A Formal Approach to Conformance Testing

Jan Tretmans

A Formal Approach to Conformance Testing

CIP–GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Tretmans, Gerrit Jan

A formal approach to conformance testing / Gerrit Jan
Tretmans. – [S.l. : s.n.]. – Ill.
Proefschrift Enschede. – Met lit. opg., reg.
ISBN 90–9005643–2
Trefw.: communicatiesystemen ; tests.

A Formal Approach to Conformance Testing

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente
op gezag van de rector magnificus
prof. dr. Th.J.A. Popma
volgens besluit van het College van Dekanen
in het openbaar te verdedigen
op donderdag 10 december 1992 te 13.15 uur

door

Gerrit Jan Tretmans

geboren op 27 augustus 1962
te Hengelo Ov.

Dit proefschrift is goedgekeurd door de promotor
prof. dr. H. Brinksma

Abstract

In order to assure successful communication between computer systems from different manufacturers, standardized communication protocols are being developed and specified. As a next step implementations of these protocols are needed that conform to these specifications. Testing is a way to check correctness of protocol implementations with respect to their specifications. This activity is known as *protocol conformance testing*.

This thesis deals with a formal approach to protocol conformance testing. Testing is performed based on a formal specification of the protocol. The final aim is to obtain methods for the (automatic) derivation of useful sets of tests from formal specifications. The derived tests should be provably correct, which means that they should not detect errors in correct implementations. Moreover, the derived tests should be meaningful: erroneous implementations should be detected with a high probability. An important aspect is a formal definition of what constitutes correctness, i.e. when does a protocol implementation conform to a protocol specification.

Starting points for this thesis are the current, informal approach to conformance testing as it is described in the international standard ISO IS-9646 "OSI Conformance Testing Methodology and Framework", and the specification formalisms for distributed systems based on labelled transition systems and process algebra. The most important concepts of the standard ISO IS-9646, and of the specification formalisms used are introduced in chapter 1.

Chapter 2 presents a framework for conformance testing. It is derived by giving a formal interpretation to the most important concepts from the standard ISO IS-9646, such as conformance requirement, the meaning of conformance, test purpose, test method, and different kinds of tests. This interpretation is shown to lead naturally to a definition of conformance as a (preorder) relation on the specification formalism. Such a relation is called an *implementation relation*.

In chapter 3 the framework is elaborated with existing implementation relations for labelled transition systems. The relations are introduced using the principle of observations: the behaviour of an implementation is correct if all observations made of the implementation by an environment, can be explained from the behaviour of the specification.

In chapter 4 test derivation algorithms are developed for one particular implementation

relation, the relation **conf**. These algorithms can be used to derive tests from a labelled transition system specification, which are complete and correct: an implementation is correct according to **conf** if and only if all tests are successful. The test derivation algorithms take labelled transition systems, and consequently they can also be used to derive tests from a behaviour expression in a formal language for which the semantics is defined by a labelled transition system. A problem appears if this labelled transition system is infinite in number of branches or number of states. Infinity prevents the algorithms from being implemented. Therefore the algorithms are transformed into algorithms that work on behaviour expressions. For a simple class of behaviour expressions rules are given with which tests can be derived compositionally from these behaviour expressions.

The implementation relations of chapter 3 and the test derivation algorithms of chapter 4 assume synchronous communication between the tester and the implementation, as can be modelled by the parallel synchronization operator on labelled transition systems. Chapter 5 shows that this theory is not applicable when the tester and the implementation communicate via a FIFO buffer. Tests derived for synchronous communication cannot detect all erroneous implementations, while correct implementations are wrongly considered to be erroneous. To demonstrate this a *queue-operator* on labelled transition systems is defined to formalize asynchronous communication between the tester and the implementation. This queue operator models two queues, one for input and one for output. A system that communicates asynchronously with its environment is called a *queue-context*. It is shown that the behaviour of queue contexts is characterized by two sets of sequences of actions (*traces*): sequences that can be performed by a queue context, and sequences that result in a state in which no output is possible. Implementation relations for asynchronous communication are derived, and first steps towards test derivation algorithms for these relations are made. The derived tests are translated into TTCN, the standardized test notation from the standard ISO IS-9646. Since TTCN uses a mechanism for communication based on queues this translation makes sense, as opposed to a translation of synchronous tests.

Using the test derivation algorithms of chapters 4 and 5, large numbers of tests can be derived. The reduction of the number of tests to an amount that can be handled economically and practically is called *test selection*. Chapter 6 studies a framework for test selection based on assigning values and costs to sets of tests. The value is related to the error detecting capability of a set of tests. The framework, which is independent from any specification formalism, is elaborated for labelled transition systems. Moreover, a test selection technique based on specification selection is presented for labelled transition systems. Instead of selecting tests from a too large set of tests, the specification is transformed, such that the tests derived from the transformed, partial specification exactly constitute a selection of the tests derived from the original specification.

The last chapter, chapter 7, presents conclusions and a few suggestions for further research: the relation between testing and verification, asynchronous communication with other kinds of contexts, extension of test selection methods to realistic specifications, the integration of values and costs, and design for testability of protocols. Finally it is

noted that the applicability and shortcomings of all presented ideas need to be validated by applying them to testing realistic protocols.

Acknowledgements

I am indebted to my promotor Ed Brinksma for his ideas, inspiration, constructive criticism, assistance in the production of this thesis, and being co-author of some of the chapters. Without his help it would have been difficult to complete this thesis.

I would like to thank Pim Kars for being my roommate, for reading and commenting almost all my drafts, some of them very immature, and for being there to answer my questions.

Louis Verhaard and Jeroen van de Lagemaat were co-authors for some parts of this thesis, for which I thank them.

Henk Eertink, Rom Langerak, Arend Rensink, Pippo Scollo, Rudie Alderden, Jan Kroon and Stan van de Burgt are thanked for commenting on earlier versions of some chapters.

I would like to thank Chris Vissers for giving me the opportunity to work in the stimulating environment of the TIOS group, and all the members of the TIOS group for many enlightning discussions and mental support.

Finally, I thank my parents for encouraging and supporting me during my whole study, and Andèle for her love, support, and understanding.

Contents

Chapter 1

Introduction

1.1 Conformance Testing and the Use of Formal Methods

The development of distributed systems, in which the computer functionality, such as processing functions, information storage, and human interaction, is distributed over different computer systems, raises the need for exchanging information between these systems. To have computer systems communicate successfully, the communication must occur according to well-defined rules. A *protocol* describes the rules with which computer systems have to comply in their communication with other computer systems. A *protocol entity* is that part of a computer system that takes care of the local responsibilities in communicating according to the protocol.

To have successful communication also among computer systems of different manufacturers, many protocols are not developed in isolation, but within groups of manufacturers and users, with the aim of standardizing such protocols. This has led for instance to the development of the OSI Reference Model for Open Systems [ISO84], which serves as a framework for a set of standards that enable computer systems to communicate. However, to assure successful communication it is not sufficient to specify and standardize communication protocols. It must also be possible to ascertain that the implementations of these protocols really *conform* to these standard protocol specifications. One way to do this is by *testing* these protocol implementations. This activity is known as *protocol conformance testing*.

In the design, specification, and analysis of protocols the use of formal, mathematically based methods increases. This makes it desirable and possible to base testing of protocol implementations on formal methods as well. This thesis deals with the use of formal methods in protocol conformance testing.

This section continues with an introduction to protocol conformance testing, a discussion of how testing fits within the development trajectory of protocols, and a rationale

for using formal methods in conformance testing. Section 1.2 gives an overview of the rest of this thesis.

1.1.1 Protocol Conformance Testing

Testing is the process of trying to find errors in a system by means of experimentation. The experimentation is usually carried out in a special environment, where normal and exceptional use is simulated. The aim of testing is to gain confidence that during normal use the system will work satisfactory: since testing of realistic systems can never be exhaustive, because systems can only be tested during a restricted period of time, testing cannot ensure complete correctness of an implementation. It can only show the presence of errors, not their absence.

Protocol conformance testing is a branch of testing where an implementation of a protocol entity is tested with respect to its specification. The aim is to gain confidence in the correct functioning of the implementation, and hence to improve the probability that the implementation will communicate successfully with its environment.

To conduct testing, experiments, or tests must be systematically devised. These tests are applied to an implementation, and the test outcomes are compared with the expected or calculated outcomes. Based on the results of the comparison a verdict can be formulated about the correctness of the implementation, which, if negative, can be used for improving the implementation.

In testing, in particular in software testing (see e.g. [Mye79, Whi87]), a distinction is made between functional testing and structural testing. *Structural testing*, also referred to as *white-box testing*, is based on the internal structure of a computer program. The aim is to exercise thoroughly the program code, e.g. by executing each statement at least once. Tests are derived from the program code. With *functional testing* externally observed functionality of a program is tested from its specification. It is also called *black-box testing*: a system is treated as a black box, whose functionality is determined by observing it, i.e. no reference is made to the internal structure of the program. The main goal is to determine whether the right (with respect to the specification) product has been built. Functional tests are derived from the specification. A prerequisite is a precise and clear specification.

Conformance testing is a kind of functional testing: an implementation of a protocol entity is solely tested with respect to its specification. Only the observable behaviour of the implementation is tested, i.e. the interactions of an implementation with its environment; no reference is made to the internal structure of the protocol implementation. In practical conformance testing the internal structure of the entity is usually not even accessible to the tester: the computer system in which the entity under test is located need not be accessible. The tester can only observe the entity by communicating with it.

Other kinds of protocol testing Since in practice it turns out that functional testing of an implementation in isolation, i.e. conformance testing, does not guarantee successful communication between systems, products are also tested in a realistic environment, for example in a model of a communication network. In this kind of testing the interaction with other computer systems can be examined in more detail. It is referred to as *interoperability testing*.

Apart from testing the functional behaviour of a protocol implementation, other kinds of testing test other aspects of a protocol, e.g. *performance testing* to measure the quantitative aspects of an implementation, *robustness testing* to examine the implementation's behaviour in an erroneous behaving environment, and *reliability testing* to test whether the implementation works correctly during a certain period of time.

Parties involved Conformance testing can be performed by different parties. First, the implementer or supplier of a product tests its product before selling it. Users of products, or their representative organizations, test products for their correct functioning. Telecommunications administrations check products before connecting them to their networks to prevent malfunctioning of a network caused by incorrectly implemented products. Finally, independent third party test laboratories can perform conformance tests for any of the previously mentioned parties. A system of accreditation allows testing laboratories to certify implementations that they have tested and judged to be conforming. Certification by accredited testing laboratories makes repeated testing by supplier, buyer, and network owner superfluous.

Standardization of conformance testing If implementations of the same (internationally) standardized protocol are tested it should not occur that different test laboratories decide differently about conformance of the same implementation. Ideally, it should not be necessary that the same product is tested more than once by different testing laboratories. This is possible if testing is based on generally accepted principles, using generally accepted tests, and leading to generally accepted test results. To achieve this the International Organization for Standardization (ISO), together with the CCITT, has developed a standard for conformance testing of Open Systems. This is the standard ISO IS-9646: 'OSI Conformance Testing Methodology and Framework' [ISO91a].

The purpose of this standard is 'to define the methodology, to provide a framework for specifying conformance test suites, and to define the procedures to be followed during testing', leading to 'comparability and wide acceptance of test results produced by different test laboratories, and thereby minimizing the need for repeated conformance testing of the same system' [ISO91a, part 1, Introduction]. The standard does not specify tests for specific protocols, but it defines a framework in which such tests should be developed, and it gives directions for the execution of such tests. The standard recommends that sets of tests, called *test suites*, be developed and standardized for all standardized protocols. A brief overview of the standard ISO IS-9646 is given in section 1.3.

1.1.2 Formal Methods

The increasing complexity and the need for specifications that are well-defined, complete, consistent, unambiguous and precise, lead to the use of formal methods in the specification of distributed systems and protocols. Using formal methods the behaviour of a distributed system is described in a language with a formally defined syntax and semantics, instead of a natural language such as English. Formal descriptions (FD) of behaviour written in a Formal Description Technique (FDT) that has a well-defined, formal mathematical model, provide precise and unambiguous specifications, allow to reason about them formally, and make it possible to analyse them using mathematical means. Moreover, the use of FDTs allows the definition and implementation of tool functions that can support the development process of complex distributed systems.

Some of the formal description techniques that are mainly intended to be applied for the specification of protocols in the context of open systems are standardized. Standardized FDTs are Estelle [ISO89a] and SDL [CCI88], of which the underlying mathematical model is based on extended finite state machines, and LOTOS [ISO89b], which is based on the theory of process algebras.

The Formal Protocol Development Trajectory

The process of developing a system based on formal methods, as advocated in e.g. [LOT92], is schematically depicted in figure 1.1. We discuss it briefly.

Informal requirements The first phase in the development process is the requirements capturing phase. The intended user is asked about her or his requirements and expectations about the system that is developed. This phase is informal, and the result consists of an overview of the user requirements, mostly written in a natural language. The informal user requirements are not guaranteed to be complete, nor consistent.

Formal specification A specification is developed from the user requirements using a formal technique. The specification should be sufficiently concrete to guarantee satisfaction of the user requirements, but sufficiently abstract not to fix irrelevant details, and to leave freedom of implementation choices. This specification being a formal object can be formally analysed for its properties, e.g. it can be formally verified that it does not possess unspecified deadlocks.

Implementation The formal specification of the specification phase is transformed in a number of implementation steps ($1 \ldots n$) into a final implementation (*implementation* n). Each implementation step consists of transforming a higher level, relatively more abstract, formal description (*implementation* i) into a lower level, more concrete, formal description (*implementation* $i + 1$). In each transformation step design decisions are made.

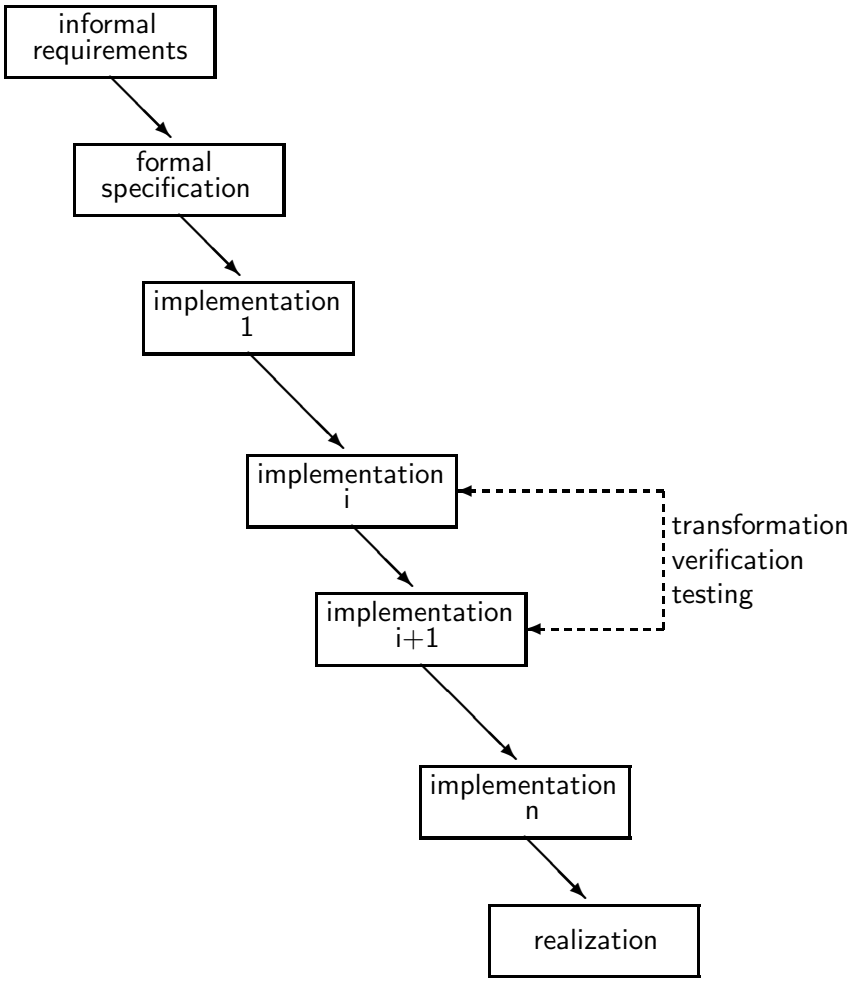


Figure 1.1: System development process.

Realization In the final step a realization is derived from the most concrete implementation (*implementation n*). While the most concrete implementation is still a formal description of the system under development, the realization is a physical system, the concrete product that is the goal of the whole development process. The realization can consist of e.g. hardware components, a single chip, or a piece of executable code.

Correctness An important aspect of going through the successive phases of the development trajectory is the correctness of the different system descriptions. This concerns all phases. When the initial ideas and user requirements have been combined and laid down in the formal specification, this specification must be analysed to check whether it indeed specifies what was intended, and whether it is internally consistent. This is referred to as *protocol design validation*. Also each implementation step, and the final realization step have to be checked for correctness with respect to their previous step, and with respect to the initial specification.

To check for correctness it must first be defined what correctness is. An observation is that a necessary condition for correctness is that successive system descriptions at least preserve the user requirements, in order not to end up with a realization that does not fulfil the user's needs.

When it has been defined what correctness is, there are different techniques to check it. To establish correctness of successive design steps we can use transformation, verification, or experimentation.

Transformation If the design step is the result of (automatically) transforming a relatively abstract description into a more concrete description, checking the correctness of the design step can be replaced by checking correctness of the algorithm or the transformation tool used. An example is the use of a compiler, e.g. to transform the last implementation n into executable code.

Verification Verification consists of proving formally, i.e. by mathematical means, the correctness of one formal description with respect to another. It is based on the formal semantics of the formal description technique used.

Experimentation Correctness can be made plausible by applying experiments or *tests* to system descriptions at different levels, and comparing the experimentation results. To experiment with an abstract formal description it is necessary that the description is executable, e.g. in the form of a simulation model of the system.

If we wish to check correctness of two given descriptions with respect to each other, verification gives the highest degree of certainty. However, this is not always possible. The first, and fundamental reason is that the initial user requirements and the final realization are no formal objects. User requirements usually consist of a natural language

document, and the realization may be a physical product. These cannot be the object of formal verification, since for verification it is necessary that the system descriptions that are compared are both formal objects, amenable to mathematical proof. The second reason is that distributed systems tend to be very large and complex, which makes that verification is not feasible without powerful proof assistant tools. Such tools are not available. A third reason applies specifically in the context of open systems, where conformance of an implementation to a specification is certified by an independent, accredited test laboratory. Implementers of protocols normally do not allow the test laboratories access to the implementation details of their products to verify correctness.

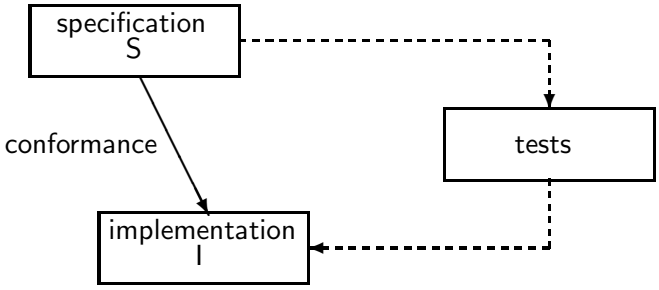


Figure 1.2: Relative specification and implementation.

The alternative to verification in these cases is experimentation. If experiments are aimed at checking correctness of an implementation or realization with respect to a more abstract, formal description, where the experiments are systematically derived from the abstract formal description, we refer to it as *conformance testing*. In general, by the term *specification* we understand the relatively abstract, formal description, and by the term *implementation* the more concrete description or physical product. The general problem of conformance testing is then depicted in figure 1.2: the conformance of an implementation I is checked against the formal specification S by deriving tests from S and applying them to I. The specification S stands for the formal specification or any of the implementations $1 \ldots n$ in figure 1.1; the implementation I stands for any of the implementations $1 \ldots n$, or the realization.

1.1.3 Formal Methods in Protocol Conformance Testing

Formal methods in protocol conformance testing concern testing of protocol implementations for conformance with respect to their specifications, where the specification is given by means of a formal description, and where testing is based on a formal definition of what constitutes conformance. Correctness of the specification is assumed, and is not considered as part of conformance testing.

On the one hand the use of formal methods in conformance testing is a consequence of specifying protocols by means of formal descriptions, thus creating a need to assess conformance based on these formal descriptions. On the other hand the use of

formal methods has benefits in conformance testing itself. First, concepts of conformance testing can be defined formally, and thus more precisely. Secondly, the use of FDTs (Formal Description Techniques) makes it possible to do *formal test validation*, i.e. checking whether a test really tests what it is intended for. Thirdly, the use of formalisms allows the definition of test generation algorithms. This will obviate the need for manual generation of tests for each protocol specification, thus avoiding the main bottle-neck of the current conformance testing methodology. Using formal methods, algorithms for test generation can be standardized instead of standardizing a test suite for each standardized protocol. This will also have large benefits in test suite maintenance, e.g. in case of modification of a protocol. Finally, the use of formal methods allows the automatic interpretation of test outcomes, thus accelerating the assessment of conformance.

Standardization of formal methods in conformance testing The standard ISO9646 [ISO91a] makes no strong assumptions about the form of the protocol specifications, but the presented methodology is intended mainly for specifications written in a natural language. When FDTs are used, it will be necessary to examine the consequences for the standardization of conformance testing. For that purpose a joint project within ISO and CCITT was launched: 'Formal Methods in Conformance Testing' (ISO/IEC JTC 1/SC 21/WG 1 Project 54, CCITT Question 10/X [ISO91b]). The scope of this project is 'to define a general methodology on how to perform conformance testing of a protocol implementation given a formal specification of a protocol standard' [ISO91b, section 1].

1.2 Overview of the Thesis

Our goal is to develop a formal approach to conformance testing. Starting points are the informal testing methodology of the standard ISO9646, which is the basis of current practice in conformance testing, and the formalism of labelled transition systems, which is suitable for the formalization of protocol concepts, and which forms the underlying model for a number of formal description techniques. These two starting points are briefly outlined in the two remaining sections of this introductory chapter. Section 1.3 has been published as [TL90].

Chapter 2 formalizes the testing methodology of ISO9646, giving a formal interpretation to the most important concepts in this standard. From this formal interpretation a formal framework for testing is developed. It is illustrated using the formalism of labelled transition systems. Chapter 2 has been published as [TKB92].

Chapter 3 applies the testing framework to labelled transition systems, and studies the question of what constitutes conformance in this model. For the purpose of testing on the basis of labelled transitions systems chapter 3 introduces two notations for tests: labelled transition systems themselves, and a special class of deterministic labelled transition systems.

Using the test notations and the definition of conformance introduced in chapter 3, chapter 4 elaborates on algorithms for the derivation tests from labelled transition system specifications.

While chapters 3 and 4 assume that execution of test cases is performed with direct, synchronous communication between test case and implementation, chapter 5 shows that these results are invalid when the communication is asynchronous. Asynchronous communication is formalized in the realm of labelled transition systems by means of a pair of queues. Communication between tester and implementation via such queues, correctness of implementations that communicate via queues, and test case derivation are elaborated. This work has been published as [TV92, VTKB92].

All test derivation algorithms turn out to produce infinitely many tests for any realistic specification. Chapter 6 discusses test selection to reduce the number of derived tests, both in a general setting, and applied to labelled transition systems. This work is based on [BTV91].

Chapter 7 contains the conclusions and discusses some open problems.

Some mathematical preliminaries and notations are presented in appendix A. Proofs of all propositions and theorems are contained in appendix B.

1.3 Overview of ISO IS-9646

The current practice of protocol conformance testing is based on the standard ISO IS-9646, 'OSI Conformance Testing Methodology and Framework' [ISO91a, Ray87]. This standard defines a methodology and framework for protocol conformance testing assuming that protocols are specified using a natural language. It was originally developed for OSI protocols, but it is also used for testing other kinds protocols, e.g. ISDN protocols. The standard consists of five parts, each defining an aspect of conformance testing:

- part 1 is an introduction and deals with the general concepts;
- part 2 describes the process of abstract test suite specification;
- part 3 defines the test notation TTCN;
- part 4 deals with the execution of tests;
- part 5 describes the requirements on test laboratories and their clients during the conformance assessment process.

This section gives a brief overview of ISO9646. Most attention is spent on parts 1 and 2, i.e. the generation and specification of test suites, since the next chapters concentrate mainly on these aspects of conformance testing.

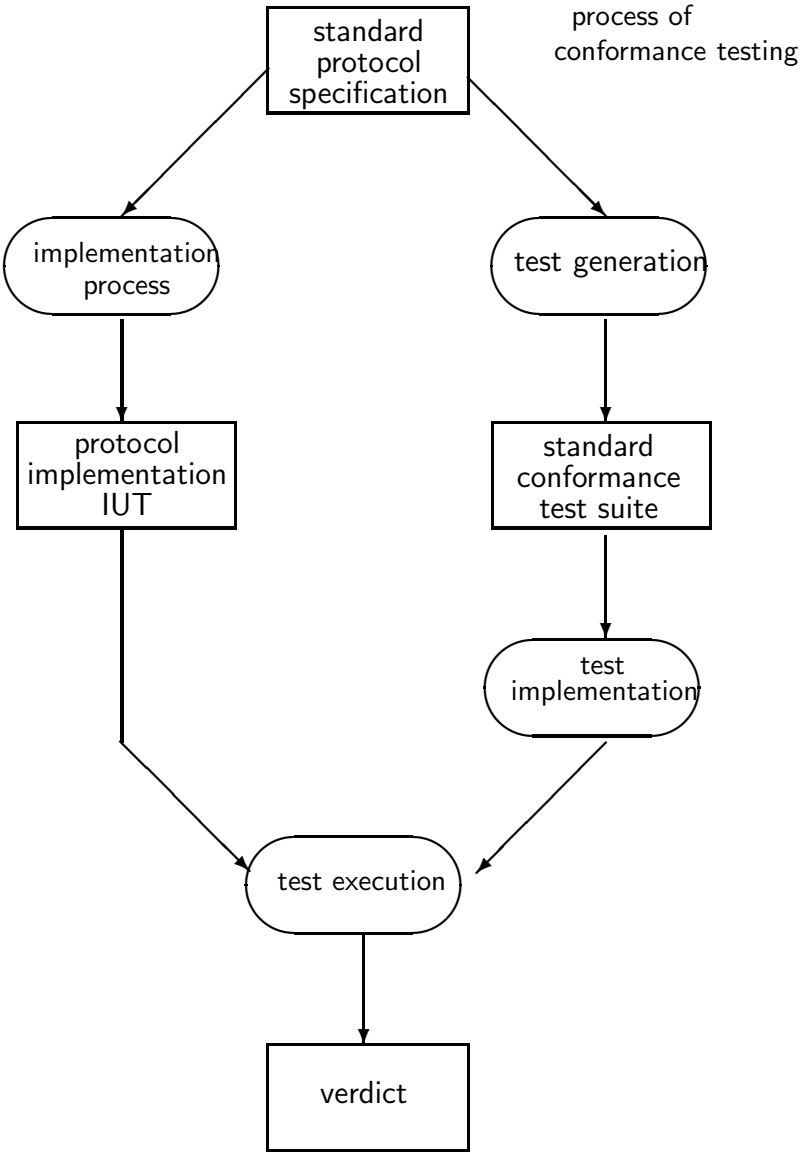


Figure 1.3: Global overview of the conformance testing process.

1.3.1 The Conformance Testing Process

In the process of conformance testing three phases are distinguished [ISO91a, part 1, section 1.3]. They are depicted in figure 1.3, together with the activity of protocol implementation. The first phase is the specification of an *abstract test suite* for a particular (OSI) protocol. We refer to it by *test generation* or *test derivation*. This test suite is abstract in the sense that tests are developed independently of any implementation. It is intended that abstract test suites of standardized protocols are standardized themselves. The second phase consists of the realization of the means of executing specific test suites. It is referred to by *test implementation*. The abstract test cases of the abstract test suite are transformed into executable tests that can be executed or interpreted on a real testing device or test system. The peculiarities of the testing environment and the implementation, which during testing is called *IUT* (Implementation Under Test), are taken into account. The last phase is the *test execution*. The implemented test cases are executed with a particular IUT and the resulting behaviour of the IUT is observed. This leads to the assignment of a verdict about conformance of the IUT with respect to the standard protocol specification. The results of the test execution are documented in the *protocol conformance test report* (PCTR).

In the next subsections these phases are described in more detail. This leads to a more detailed view of the conformance testing process given in figure 1.4.

1.3.2 A Conforming Implementation

Before an implementation can be tested for conformance it must be defined what it means for an implementation to conform to its specification. The definition of what constitutes a conforming implementation determines what should be tested. ISO9646 states that a system 'exhibits conformance if it complies with the conformance requirements of the applicable ... standard' [ISO91a, part 1, section 5.1]. This means that a correct implementation is one which satisfies all conformance requirements, and that these conformance requirements must be mentioned explicitly in the protocol standard. Conformance requirements express what a conforming implementation shall do (positively specified requirements), and what it shall not do (negatively specified requirements).

A complication arises by the fact that a protocol standard does not uniquely specify one protocol, but a class of protocols. Most standards leave open a lot of options, which may or may not be implemented in a particular protocol implementation, but which, if implemented, must be implemented correctly. An implementer selects a set of options for implementation. All implemented options of a specific protocol implementation are listed by the implementer in the *PICS*, the *Protocol Implementation Conformance Statement*, so that the tester knows which options have to be tested. To assist in producing the PICS a *PICS proforma* is attached to the protocol standard. This is a questionnaire in which all possibilities for the selection of options are enumerated.

Restrictions on the selection of options are given in the *static conformance requirements* of a standard. They define requirements on the minimum capabilities that an implementation is to provide, and on the combination and consistency of different options.

Example 1.1
In the ISO/OSI Transport Protocol [ISO86] five classes (0 .. 4) are distinguished. In a particular implementation not all classes need be implemented. However, the choice is not completely free, e.g. if class 4 is implemented also class 2 must be implemented. Such a restriction is recorded in the protocol standard as part of the static conformance requirements. In the PICS the implemented classes of a particular implementation are documented.
□

The main part of a protocol standard consists of *dynamic conformance requirements*. They define requirements on the observable behaviour of implementations in the communication with their environment. They concern the allowed orderings of observable events, such as sending and receiving of PDUs (protocol data units) and ASPs (abstract service primitives), the coding of information in the PDUs, and the relation between information content of different PDUs.

Example 1.2
A dynamic conformance requirement of the ISO/OSI Transport Protocol is the requirement that after receiving a T-PDU-connect-request from the peer entity either the user of the Transport entity is notified by means of a T-SP-connect-indication service-primitive, or a T-PDU-disconnect-request is sent to the peer entity.
□

Summarizing, the definition of a conforming implementation is [ISO91a, part 1, sections 3.4.10 and 5.6]:

> 'A conforming implementation is one which satisfies both static and dynamic conformance requirements, consistent with the capabilities stated in the PICS.'

Conformance testing consists of checking whether an IUT satisfies all static and dynamic conformance requirements. For the static conformance requirements this means a reviewing process of the PICS delivered with the IUT. This is referred to as the *static conformance review*. For the dynamic conformance requirements this means running a number of tests against the IUT. One test is referred to as a *test case*. A *test suite* is a complete set of test cases, i.e. a set that tests all dynamic conformance requirements.

1.3.3 Test Generation

The first phase of the conformance testing process is *test generation*. It consists of systematically deriving test cases from a protocol specification. The goal is to develop an abstract test suite, i.e. a specification of a test suite that is implementation independent,

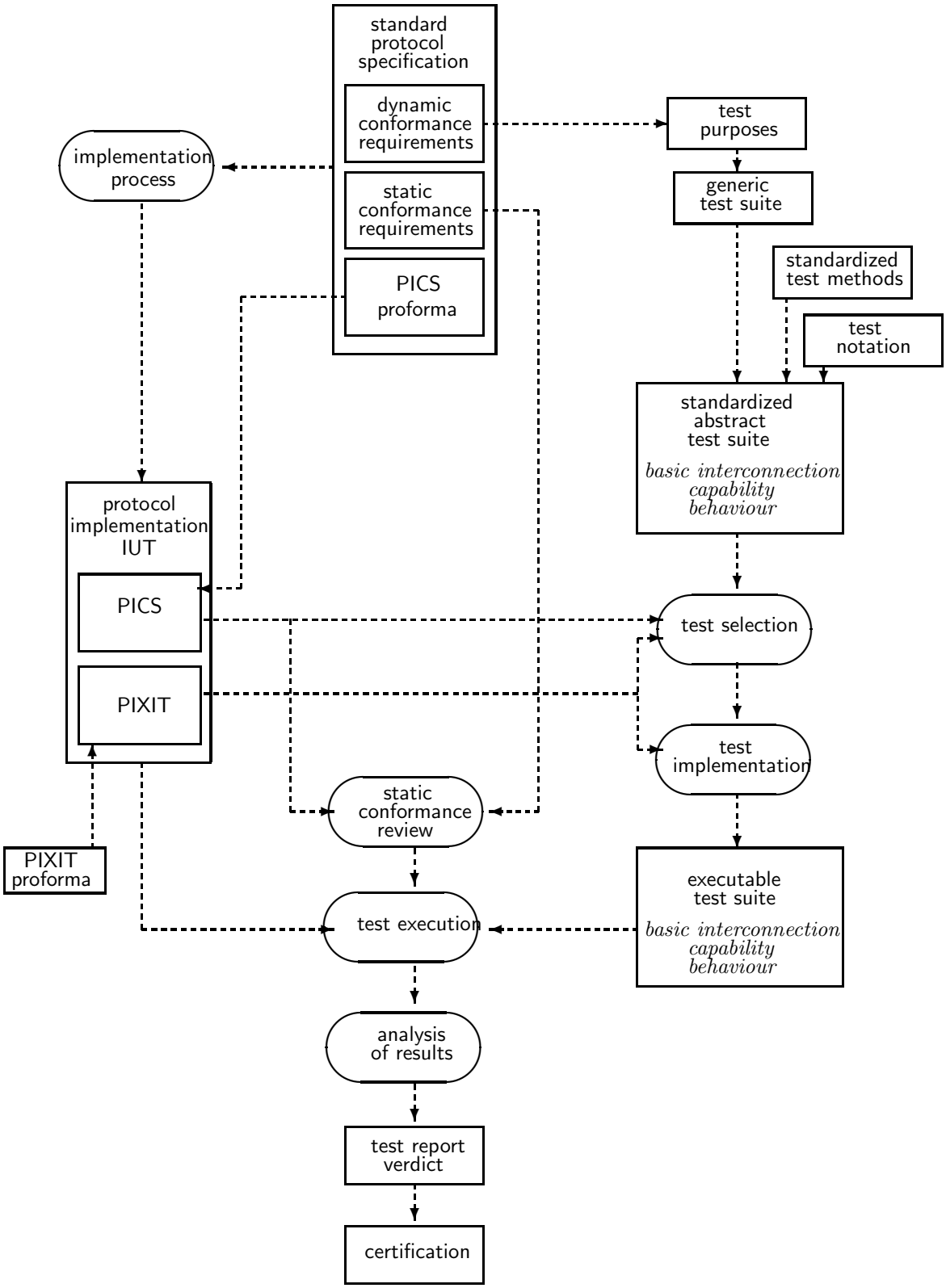


Figure 1.4: Detailed overview of the conformance testing process.

specified in a well-defined test notation language, suitable for standardization, and testing all aspects of the protocol in sufficient detail. Since the relevance of a protocol specification with respect to conformance testing is its set of conformance requirements, and since the static conformance requirements are checked by reviewing the PICS, this means that the set of the dynamic conformance requirements in a protocol standard is the starting point for test generation.

Test cases are derived systematically from the dynamic conformance requirements in a multi-step procedure. In the first step, one or more *test purposes* are derived for each conformance requirement. A test purpose is a precise description of what is going to be tested in order to decide about the satisfaction of a particular conformance requirement. As the next step it is recommended to derive a *generic test case* for each test purpose. A generic test case is an operationalization of a test purpose, in which the actions necessary to achieve the test purpose are described on a high level, without considering a test method or the environment in which the actual testing will be done. The last step is the derivation of an *abstract test case* for each generic test case. In this step a choice is made for a particular *test method*, and the restrictions implied by the environment in which testing will be carried out are taken into account.

Test methods

A protocol standard specifies the behaviour of a protocol entity at the upper and lower access points of the protocol ((N)-SAP en (N-1)-SAP). Hence the ideal points to test the entity are these SAPs. However, these SAPs are not always directly accessible to the tester. The points where the tester controls and observes the IUT are called the *Points of Control and Observation* (*PCO*). PCOs may, but need not coincide with the boundaries of the IUT. Normally there are two PCOs, one corresponding with the upper access point of the IUT, and one with the lower access point. A similar conceptual separation is made for the tester. The part of the tester that controls and observes the PCO connected to the upper access point is called the *Upper Tester* (UT). The part that controls and observes the PCO connected to the lower access point is called the *Lower Tester* (LT).

A *test method* defines a model for the accessibility of the IUT to the tester in terms of PCOs and their place within the OSI reference model [ISO84]. Aspects that can be distinguished are:

- existence of PCOs: if one of the access points is not accessible at all there is no PCO for that access point;
- whether there are other protocol layers between the PCO and the access point, and the kind of events that are communicated (ASPs or PDUs);
- the positioning of the PCOs in the same computer system as the IUT, called the *System Under Test* (SUT);
- the internal functioning of the tester in terms of the distribution of testing functions over LT and UT, and the rules that define their coordination: the *test coordination*

procedures.

By varying these aspects different test methods are obtained. Some have been identified and standardized in ISO9646 [ISO91a, part 2, section 12] for use in standardized abstract test suites. In these standardized test methods the lower access point of the IUT is always accessible via an underlying service; the upper access point may be hidden. Standardized test methods are the Local Single-layer test method (LS-method), the Distributed Single-layer test method (DS-method), the Coordinated Single-layer test method (CS-method), and the Remote Single-layer test method (RS-method). Figure 1.5 shows the DS-method as an example. There are two PCOs. An example of a test method with one PCO is the RS-method: in the RS-method there is no upper tester.

A standardized abstract test suite refers to a particular test method, choosing the most appropriate one.

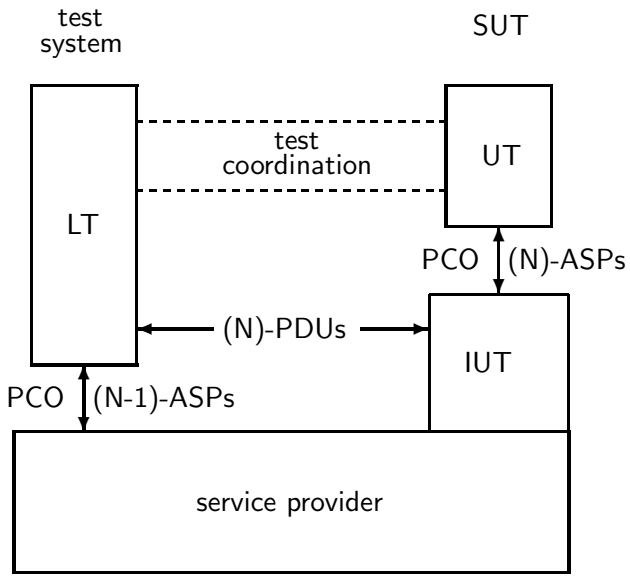


Figure 1.5: The distributed test method.

The four test methods that were mentioned, can be used in variations where the IUT consists of more than one subsequent protocol layers. These layers can be tested as a whole (multi-layer testing), or one layer can be tested embedded in the other layers (embedded testing). The test methods are LM, CM, DM and RM (Local Multi-layer, etc.), and LSE, CSE, DSE and RSE (Local Single-layer Embedded, etc).

Test notation

Since abstract test suites are standardized, they must be specified in a *test notation* that is well-defined, independent of any implementation, and generally accepted. IS-

9646 recommends the semi-formal language *TTCN*, the *Tree and Tabular Combined Notation*, which is defined in [ISO91a, part 3].

In TTCN the behaviour of test cases is specified by sequences of input and output events that occur at the PCOs. A sequence can have different alternatives, where different subsequent behaviours can be chosen, e.g. depending on output produced by the IUT, the expiration of timers, or values of internal parameters of the tester. Successive events are indicated by increasing the level of indentation, alternative events have the same indentation. A sequence ends with the specification of the verdict that is assigned when the execution of the sequence ends. The verdicts in the different possible alternative behaviours differ. Some alternatives will describe correct behaviour, ending with a positive verdict, while other alternatives describe erroneous behaviour, ending with a negative verdict.

TTCN is defined in such a way that automatic execution is feasible. A simplified example of a TTCN behaviour is presented in figure 1.6.

Test Case Dynamic Behaviour		
Test Case Name:	Conn_Estab	
Group:	transport/connection	
Purpose:	Check connection establishment with remote initiative	
behaviour	constraints	verdict
+ preamble		
LT ! T-PDU-connect-request		
UT ? T-SP-connect-indication		
UT ! T-SP-connect-response		
LT ? T-PDU-connect-confirm		**pass**
OTHERWISE		**fail**
LT ? T-PDU-disconnect-request		**inconclusive**
OTHERWISE		**fail**

Figure 1.6: A simplified TTCN example.

Classification of tests

Tests can be classified according to the extent to which they give an indication of conformance. The following distinction is made:

- basic interconnection tests
- capability tests
- behaviour tests
- conformance resolution tests

The classification is applicable to generic, abstract and the executable tests, which will be discussed in section 1.3.4.

Basic interconnection tests are used to guarantee a basic level of interconnection between the tester and the IUT. Their main purpose is economical: before an expensive test environment is developed first some basic functions of the IUT are checked, e.g. the establishment of a connection between the tester and the IUT.

Capability tests serve to verify the compliance between the implemented options and the options stated in the PICS.

Behaviour tests constitute the main part of a test suite. They test the dynamic conformance requirements of a protocol standard in full detail within the limits of technical and economical feasibility. They are the basis for the final verdict about conformance.

Conformance resolution tests do not belong to the actual conformance tests. They form supplementary tests that can be used to do extra testing if problems are encountered, or to trace errors. These tests have a heuristic nature, they are not standardized, and they cannot be used as a basis for the final verdict.

Hierarchical structuring of tests

A test suite is a complete set of tests for conformance testing of a particular protocol. Elements of a test suite are tests, or test cases. A test case specifies one experiment, related to one test purpose and to one conformance requirement. Related test cases can be grouped into test groups with corresponding test group objective. Grouping can occur at different levels.

Within a test case test stepsa and test events can be distinguished. A test event is one interaction at a PCO, e.g sending or receiving one PDU. A test step groups successive test events. An example of a test step is a preamble: a sequence of events that brings the IUT in a state from which the body of the test case that tests the test purpose can be tested. Analogously the postamble test step brings the IUT back to a specified state, e.g. the initial state, after the main part of a test case has been executed.

The hierarchical structuring is applicable to all levels of test cases. Also conformance requirements and test purposes can be grouped.

1.3.4 Test Implementation

Starting point for test implementation is the (standardized) abstract test suite. The abstract test suite is specified independently of any real testing device. In the test implementation phase it is transformed into an *executable test suite*, i.e. a test suite which can be run on a specific testing device with a specific IUT.

Before starting to implement, a selection from the abstract test suite must be made. The abstract test suite contains all possible tests for a particular protocol, for all possible options. It does not make sense to test for options that are not implemented according

to the PICS. Therefore the tests relevant to the IUT are selected based on the PICS. In ISO9646 this is called *test selection*[1].

The PICS contains protocol dependent information. To derive executable tests this is insufficient; also information about the IUT and its environment must be supplied. Such information is called PIXIT (Protocol Implementation eXtra Information for Testing). The PIXIT may contain address information of the IUT, or parameter and timer values which are necessary to implement the test suite. The PIXIT, like the PICS, is supplied by the supplier of the IUT to the testing laboratory. To guide production of the PIXIT the testing laboratory provides a *PIXIT proforma*.

The selected and implemented test cases with parameter values according to the PIXIT form the executable test suite, which can be executed on a real tester or test system. During implementation care must be taken that the tests are implemented correctly, according to the semantics of the test notation used for the specification of the abstract test suite.

1.3.5 Test Execution

During the test execution phase a specific IUT is actually tested leading to a verdict about conformance of the IUT. The first step consists of the static conformance review: the PICS of the IUT is checked for internal consistency and compared with the static conformance requirements of the standard. The second step consists of executing the executable test cases on a real tester. The reactions of the IUT are observed and compared with the reactions specified in the test case. For each test case a verdict is assigned. The verdict is either *pass*, *fail*, or *inconclusive*. *Pass* indicates that the test was executed successfully, and that the goal expressed in the corresponding test purpose was achieved. *Fail* indicates that the implementation does not conform to the specification with respect to the given test purpose. *Inconclusive* indicates that no evidence of non-conformance was found, but that the test purpose was not achieved.

Example 1.3
Suppose the test purpose in the TTCN example in figure 1.6 is to check the correct connection establishment of the Transport Protocol, by testing the sequence of actions T-PDU-connect-request, T-SP-connect-indication, T-SP-connect-response, T-PDU-connect-confirm. If the IUT reacts with T-PDU-disconnect-request after having received T-PDU-connect-request, the verdict *inconclusive* is assigned: this behaviour is allowed according to the Transport standard, but the the verdict *pass* cannot be assigned since the test purpose was not achieved.
□

Finally, the results of the static conformance review and the verdicts of all test cases are combined, leading to a verdict about conformance of the IUT with respect to the protocol specification. Normally the final verdict is *pass* if and only if no individual test

[1]Note that the notion of 'test selection' that we use, is different. It is discussed in chapter 6.

resulted in the verdict *fail*. All results, including the final verdict, are documented in the PCTR (Protocol Conformance Test Report).

1.4 Labelled Transition Systems

The formalism of *labelled transition systems* is used for modelling the behaviour of processes, systems, and components. Labelled transition systems serve as a semantic model for a number of specification languages, e.g. CCS [Mil80, Mil89], CSP [Hoa85], ACP [BK85], and LOTOS [BB87, ISO89b].

Definition 1.4
A *labelled transition system* is a 4-tuple $\langle S, L, T, s_0 \rangle$ with

- S is a (countable) non-empty set of *states*;
- L is a (countable) set of *observable actions*;
- $T \subseteq S \times (L \cup \{\tau\}) \times S$ is the *transition relation*;
- $s_0 \in S$ is the *initial state*.

□

The labels in L represent the observable interactions of a system; the special label $\tau \notin L$ represents an unobservable, internal action. Table 1.1 introduces some notation for labelled transition systems ($a \cdot b$ is the concatenation of a and b, see appendix A).

In the following we assume a universe of observable actions L, and we denote the class of all labelled transition systems over L by $\mathcal{LTS}$. Moreover, we restrict $\mathcal{LTS}$ to labelled transition systems that are *strongly converging*, i.e. ones with no infinite sequence of internal actions.

1.4.1 Representation

Labelled transition systems are a suitable formalism to model distributed systems. Simple systems can be represented by (action-) trees or graphs, where nodes represent states, and edges labelled with actions, represent transitions.

Example 1.5
Figure 1.7 gives an example of a action tree representing the labelled transition system

$$\begin{array}{ll} \langle & \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8\}, \\ & \{\mathit{shilling}, \mathit{coffee\text{-}button}, \mathit{coffee}, \mathit{tea\text{-}button}, \mathit{tea}\}, \\ & \{\langle s_0, \mathit{shilling}, s_1\rangle, \langle s_0, \mathit{shilling}, s_2\rangle, \langle s_1, \mathit{coffee\text{-}button}, s_3\rangle, \langle s_1, \mathit{tea\text{-}button}, s_4\rangle, \\ & \ \langle s_2, \mathit{coffee\text{-}button}, s_5\rangle, \langle s_3, \mathit{coffee}, s_6\rangle, \langle s_4, \mathit{tea}, s_7\rangle, \langle s_5, \mathit{coffee}, s_8\rangle\}, \\ & s_0 \ \rangle \end{array}$$

This labelled transition system models a vending machine, supplying *coffee* and *tea*. The machine specifies as the first action insertion of a *shilling*, after which the machine

notation	**meaning**
$B \xrightarrow{\mu} C$	$(B, \mu, C) \in T$
$B \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} C$	$\exists B_0 \ldots B_n : B = B_0 \xrightarrow{\mu_1} B_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} B_n = C$
$B \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n}$	$\exists C : B \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} C$
$B \not\xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n}$	$\neg\exists C : B \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} C$
$B \stackrel{\epsilon}{\Longrightarrow} C$	$B = C$ or $B \xrightarrow{\tau \cdot \tau \cdot \ldots \cdot \tau} C$
$B \stackrel{a}{\Longrightarrow} C$	$\exists B_1, B_2 : B \stackrel{\epsilon}{\Longrightarrow} B_1 \xrightarrow{a} B_2 \stackrel{\epsilon}{\Longrightarrow} C, \ \ a \in L$
$B \stackrel{a_1 \cdot \ldots \cdot a_n}{\Longrightarrow} C$	$\exists B_0 \ldots B_n : B = B_0 \stackrel{a_1}{\Longrightarrow} B_1 \stackrel{a_2}{\Longrightarrow} \ldots \stackrel{a_n}{\Longrightarrow} B_n = C$
$B \stackrel{a_1 \cdot \ldots \cdot a_n}{\Longrightarrow}$	$\exists C : B \stackrel{a_1 \cdot \ldots \cdot a_n}{\Longrightarrow} C$
$B \not\stackrel{a_1 \cdot \ldots \cdot a_n}{\Longrightarrow}$	$\neg\exists C : B \stackrel{a_1 \cdot \ldots \cdot a_n}{\Longrightarrow} C$
$B \stackrel{a_1 \cdot \ldots \cdot a_n}{\Longrightarrow} C$	$(n = 0$ and $B = C)$ or $(\exists B' : B \stackrel{a_1 \cdot \ldots \cdot a_{n-1}}{\Longrightarrow} B'$ and $B' \xrightarrow{a_n} C)$

Table 1.1: Notation for labelled transition systems.

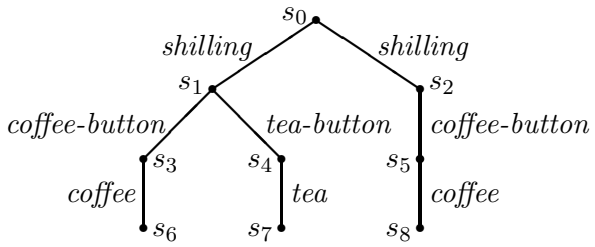


Figure 1.7: Action tree of a vending machine.

moves nondeterministically to either s_1 or s_2. In s_1 there is a choice between the actions *coffee-button* and *tea-button*; in s_2 only the *coffee-button* can be pushed. The action *coffee-button* is followed by *coffee*; the action *tea-button* by *tea*.

□

For more complex systems such as protocols, a more sophisticated way of representation than graphs or trees is needed: a language that allows concise representations of (infinite) labelled transition systems. We use the language of definition 1.6, which is inspired by the formal description technique LOTOS [BB87, ISO89b]. The operational semantics of an expression in this language, a *behaviour expression*, is given by a labelled transition system. It is defined in definition 1.7.

Definition 1.6
A *behaviour expression* B is defined by the syntax:

$$B \;\; =_{def} \;\; \mathbf{stop} \;\mid\; a;B \;\mid\; \mathbf{i};B \;\mid\; B \Box B \;\mid\; B|[G]|B \;\mid\; \Sigma\mathcal{B}$$

with $a \in L$, the universe of observable actions, $G \subseteq L$, and $\mathcal{B}$ a set of behaviour expressions.

$\mathcal{BEX}$ is the set of all behaviour expressions, i.e. the language of behaviour expressions. We use $\|$ as abbreviation for $|[L]|$, and $|||$ as abbreviation for $|[\emptyset]|$.

The priority is such that ';' binds stronger than '$\Box$', which in turn binds stronger than '$|[G]|$'.

□

Definition 1.7
The operational semantics of the behaviour expression $B \in \mathcal{BEX}$ is the labelled transition system $\ell ts(B)$, defined by

$$\ell ts(B) \;\; =_{def} \;\; \langle\, \mathcal{BEX},\; L,\; T_{\mathcal{BEX}},\; B \,\rangle$$

where $T_{\mathcal{BEX}} \subseteq \mathcal{BEX} \times (L \cup \{\tau\}) \times \mathcal{BEX}$ is the smallest relation satisfying

$B \xrightarrow{\mu} C \quad =_{def}$

if	$B = a;C$	then	$\mu = a$,
if	$B = \mathbf{i};C$	then	$\mu = \tau$,
if	$B = B_1 \Box B_2$	then	$B_1 \xrightarrow{\mu} C$ or $B_2 \xrightarrow{\mu} C$,
if	$B = B_1 \|[G]\| B_2$	then	$(\, B_1 \xrightarrow{\mu} B_1',\ \mu \notin G$ and $C = B_1' \|[G]\| B_2 \,)$ or
			$(\, B_2 \xrightarrow{\mu} B_2',\ \mu \notin G$ and $C = B_1 \|[G]\| B_2' \,)$ or
			$(\, B_1 \xrightarrow{a} B_1',\ B_2 \xrightarrow{a} B_2',\ a \in G$ and $C = B_1' \|[G]\| B_2' \,)$,
if	$B = \Sigma\mathcal{B}$	then	$\exists B' \in \mathcal{B} : B' \xrightarrow{\mu} C$.

□

In the next chapters we will not rigidly distinguish between B as a behaviour expression, B as the initial state of its semantics $\ell ts(B)$, and the semantics itself. A process is identified with the labelled transition system modelling that process, with the behaviour expression B representing the labelled transition system, and with its initial state.

Example 1.8
The coffee-machine in example 1.5 can be represented by the behaviour expression

$$\begin{array}{ll} & \mathit{shilling}; (\mathit{coffee\text{-}button}; \mathit{coffee}; \mathbf{stop} \;\Box\; \mathit{tea\text{-}button}; \mathit{tea}; \mathbf{stop}) \\ \Box & \mathit{shilling}; \mathit{coffee\text{-}button}; \mathit{coffee}; \mathbf{stop} \end{array}$$

Other expressions define 'the same' labelled transition system, e.g. the following expression. What is 'the same' is elaborated in section 1.4.3.

$$\begin{array}{l} \quad (\quad \mathit{shilling}; (\mathit{coffee\text{-}button}; \mathbf{stop} \;\Box\; \mathit{tea\text{-}button}; \mathbf{stop}) \\ \quad\; \Box \mathit{shilling}; \mathit{coffee\text{-}button}; \mathbf{stop}\ ) \\ |[\mathit{coffee\text{-}button}, \mathit{tea\text{-}button}]| \\ \quad (\mathit{coffee\text{-}button}; \mathit{coffee}; \mathbf{stop} ||| \mathit{tea\text{-}button}; \mathit{tea}; \mathbf{stop}) \end{array}$$

and

$$\begin{array}{l} \Sigma\ \{\ \mathit{shilling}; \Sigma\{\mathit{coffee\text{-}button}; \mathit{coffee}; \mathbf{stop}, \mathit{tea\text{-}button}; \mathit{tea}; \mathbf{stop}\}, \\ \quad \mathit{shilling}; \mathit{coffee\text{-}button}; \mathit{coffee}; \mathbf{stop}\ \} \end{array}$$

□

1.4.2 Traces

A *trace* is a sequence of observable actions. The set of all finite traces over L is denoted by L^*, with ϵ denoting the empty sequence. The traces of a labelled transition system specification S, $\mathit{traces}(S)$, are all sequences of visible actions that S can perform. If S can perform a trace σ then S can also perform any initial part of σ, called a *prefix* of σ. The notion of prefix is formalized by the relation $\preceq \subseteq L^* \times L^*$.

Definition 1.9

1. $\mathit{traces}(S) \;=_{def}\; \{\ \sigma \in L^* \mid S \stackrel{\sigma}{\Longrightarrow} \}$
2. A trace σ_1 is a *prefix* of σ_2, $\sigma_1 \preceq \sigma_2$, if $\exists \sigma' : \sigma_1 \cdot \sigma' = \sigma_2$. Since σ' is unique we write $\sigma' = \sigma_2 \backslash \sigma_1$.

□

The relation $\preceq$ is reflexive, transitive and anti-symmetric, hence it is a partial order. Moreover, there is no infinite sequence of distinct, prefixing traces, i.e. no sequence

$$\ldots \prec \sigma_3 \prec \sigma_2 \prec \sigma_1 \prec \sigma_0$$

hence it is well-founded. More on well-foundedness in appendix A.

Proposition 1.10

1. $\langle L^*, \preceq \rangle$ is a well-founded poset.
2. $\sigma_1 \preceq \sigma_2$ and $\sigma_2 \in \mathit{traces}(S)$ imply $\sigma_1 \in \mathit{traces}(S)$
3. $\mathit{traces}(S)$ is *prefix-closed*, i.e. it is left-closed with respect to $\preceq$
4. $\mathit{min}_{\preceq}(\ \mathit{traces}(S)\ ) = \{\epsilon\}$

□

We give some additional definitions for labelled transition systems. The set $out(S)$ contains traces of length 1; S **after** σ collects all states that can be reached after having performed σ; the *derivatives* $der(S)$ combine all these states for all possible σ.

Different notions of finiteness over labelled transition systems can be defined. If the length of traces, denoted by $|\sigma|$, is bounded, a process is said to have *finite behaviour*. For *finite state* processes the number of reachable states is finite. Such processes can have traces of arbitrary length. In an *image finite* process the number of reachable states be may infinite, as long as the number of states that can be reached for any particular trace σ is finite. In a *deterministic* process the state that is reached after having performed any trace is unique. Deterministic processes do not contain τ-transitions, and the outgoing transitions of any state are uniquely labelled. Finally, a *stable* process cannot move invisibly to another state.

Definition 1.11
Let $S \in \mathcal{LTS}$, $\sigma \in L^*$:

1. $out(S) \;=_{def}\; \{\; a \in L \mid S \stackrel{a}{\Longrightarrow} \;\}$
2. S **after** $\sigma \;=_{def}\; \{\; S' \mid S \stackrel{\sigma}{\Longrightarrow} S' \;\}$
3. $der(S) \;=_{def}\; \{\; S' \mid \exists \sigma \in L^* : S \stackrel{\sigma}{\Longrightarrow} S' \;\}$
4. S has *finite behaviour* if there is $n \in \mathbf{N}$, such that $\forall \sigma \in traces(S) : |\sigma| < n$.
5. S is *finite-state* if $der(S)$ is finite.
6. S is *image-finite* if $\forall \sigma \in L^* :$ S **after** σ is finite.
7. S is *deterministic* if for all $\sigma \in L^*$, S **after** σ has at most one element. If $\sigma \in traces(S)$, then we write S **after** σ for this element.
8. S is *stable* if $S \stackrel{\tau}{\not\longrightarrow}$.

□

Example 1.12
The labelled transition system in figure 1.7 has finite behaviour, is finite-state, image-finite, and stable, but not deterministic. S **after** $shilling \cdot coffee\text{-}button = \{\; s_3, s_5 \;\}$, $out(S) = \{shilling\}$, and $der(S) = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8\}$.

□

1.4.3 Equality

Labelled transition systems are used to model the observable behaviour of distributed systems and protocols. As such they are a suitable formalization of the notion of a process. However, 'some behaviours are more equal than others': it may occur that different labelled transition systems intuitively describe the same observable behaviour, e.g. when an action τ occurs, which is assumed to be unobservable, or when a state has two equally labelled outgoing transitions to the same state. Therefore equivalence

relations have been defined based on the notion of observable behaviour, and equality of behaviour is studied with respect to equivalence classes. A lot of equivalence relations are known from literature: observation equivalence [Mil80], strong and weak bisimulation equivalence [Par81, Mil89], failure equivalence [Hoa85], testing equivalence [DNH84], failure trace equivalence [Bae86], generalized failure equivalence [Lan90], and many others. A few equivalences are mentioned here; chapters 3 and 5 discuss more of them. In particular testing equivalence, the one that is most used in the next chapters, is discussed in section 3.2.

In comparing labelled transition systems the first observation is that the names of states, nor the existence of states that cannot be reached during any execution, influence the observable behaviour. Two labelled transition systems are *isomorphic* if their reachable states can be mapped one-to-one to each other, preserving transitions and initial states. In fact we already used this with the behaviour expressions in example 1.8: the formal semantics $\ell ts(B)$ of these expressions contain the infinite set of states, consisting of all behaviour expressions.

The second observation is that equally labelled transitions to states with equivalent behaviour cannot be discerned. *Strong bisimulation equivalence* $\sim$ identifies such systems.

Weak bisimulation equivalence $\approx$ requires a relation between the reachable states of two systems that can simulate each other: if one can perform a trace σ, the other must be able to do the same, and vice versa, and the resulting states must simulate each other again.

If the *traces* of two systems are equal they are called *trace equivalent* $\approx_{tr}$.

Definition 1.13
Let $S_1, S_2 \in \mathcal{LTS}$, $S_1 = \langle \mathcal{S}_1, L, T_1, s_{0_1} \rangle$, $S_2 = \langle \mathcal{S}_2, L, T_2, s_{0_2} \rangle$:

1. S_1 and S_2 are *isomorphic*, $S_1 \equiv S_2$, if there exists a bijection $f : der(S_1) \to der(S_2)$, such that $\forall s_1, s_2 \in der(S_1)$, $\mu \in L \cup \{\tau\}$: $s_1 \xrightarrow{\mu} s_2$ iff $f(s_1) \xrightarrow{\mu} f(s_2)$, and $f(s_{0_1}) = s_{0_2}$.
2. $S_1 \sim S_2 \;=_{def}\; \exists\, \mathcal{R} \subseteq der(S_1) \times der(S_2)$, such that $\langle s_{0_1}, s_{0_2} \rangle \in \mathcal{R}$, and
   $\forall \langle s_1, s_2 \rangle \in \mathcal{R}$, $\forall \mu \in L \cup \{\tau\}$:
   $\forall s_1' :$ if $s_1 \xrightarrow{\mu} s_1'$ then $\exists s_2' : s_2 \xrightarrow{\mu} s_2'$ and $\langle s_1', s_2' \rangle \in \mathcal{R}$; and
   $\forall s_2' :$ if $s_2 \xrightarrow{\mu} s_2'$ then $\exists s_1' : s_1 \xrightarrow{\mu} s_1'$ and $\langle s_1', s_2' \rangle \in \mathcal{R}$
3. $S_1 \approx S_2 \;=_{def}\; \exists\, \mathcal{R} \subseteq der(S_1) \times der(S_2)$, such that $\langle s_{0_1}, s_{0_2} \rangle \in \mathcal{R}$, and
   $\forall \langle s_1, s_2 \rangle \in \mathcal{R}$, $\forall \sigma \in L^*$:
   $\forall s_1' :$ if $s_1 \stackrel{\sigma}{\Longrightarrow} s_1'$ then $\exists s_2' : s_2 \stackrel{\sigma}{\Longrightarrow} s_2'$ and $\langle s_1', s_2' \rangle \in \mathcal{R}$; and
   $\forall s_2' :$ if $s_2 \stackrel{\sigma}{\Longrightarrow} s_2'$ then $\exists s_1' : s_1 \stackrel{\sigma}{\Longrightarrow} s_1'$ and $\langle s_1', s_2' \rangle \in \mathcal{R}$
4. $S_1 \approx_{tr} S_2 \;=_{def}\; traces(S_1) = traces(S_2)$

□

Proposition 1.14

1. $\equiv$, $\sim$, $\approx$, and $\approx_{tr}$ are equivalences.
2. $\equiv \;\subset\; \sim \;\subset\; \approx \;\subset\; \approx_{tr}$

□

Also many non-equivalence relations over labelled transition systems can be defined. In the next chapter they will be shown to be well suited to express the relation between a specification and an implementation. Using the concepts introduced until now two prominent ones are *trace preorder* $\leq_{tr}$ and its inverse $\geq_{tr}$. Others, which are actually more important for the formalization of testing, will be introduced in chapters 3 and 5.

Definition 1.15
Let $S_1, S_2 \in \mathcal{LTS}$:

1. $S_1 \leq_{tr} S_2 \quad =_{def} \quad traces(S_1) \subseteq traces(S_2)$
2. $S_1 \geq_{tr} S_2 \quad =_{def} \quad traces(S_1) \supseteq traces(S_2)$

□

Proposition 1.16

1. $\approx_{tr} \; = \; \leq_{tr} \cap \geq_{tr}$
2. $\leq_{tr}$ and $\geq_{tr}$ are preorders

□

Chapter 2

A Formal Framework for Conformance Testing

2.1 Introduction

Current techniques for protocol conformance testing are based on the standard ISO IS-9646: 'OSI Conformance Testing Methodology and Framework' [ISO91a], which is mainly intended for specifications written in a natural language. To study formal methods in conformance testing a formal framework is needed, in which conformance testing concepts, such as conformance requirement, correctness of an implementation, test case, verdict assignment, etc. are defined in a formal setting. In this chapter such a formal framework is developed by exploring the consequences of formally specifying protocols for the testing methodology of ISO9646.

Our starting point is the description of concepts as given in ISO9646 (see section 1.3). These are interpreted assuming that the protocol specification is given by means of a formal description (FD). This leads to formal definitions of these concepts such as conformance requirement, the meaning of conformance, PICS, test purpose, generic and abstract test case, test method, verdict, and PIXIT, in section 2.2. The presentation concentrates on the process of test generation, and not all aspects of ISO9646 will be completely dealt with; for a complete formal interpretation of ISO9646 elaborations are needed. In section 2.3 the formalized concepts are related to existing notions of conformance in concurrency theory based on preorder relations. Section 2.4 illustrates the presented ideas using the specification formalism of labelled transition systems. Finally, section 2.5 sums up the main ingredients of the formal framework for use in the following chapters.

2.2 Formal Interpretation of ISO IS-9646

Considering the three phases of the conformance testing process according to ISO9646 (section 1.3.1), it is evident that the use of formal specifications has most impact in the test generation phase. This section gives an interpretation in a formal context of some concepts of ISO9646, with emphasis on this phase. The presentation follows the ISO9646 test generation methodology as described in sections 1.3.2 and 1.3.3. It considers in turn conformance requirements and the meaning of conformance, the PICS, test purposes, test cases in general, and generic and abstract test cases separately. The relation with practical test generation algorithms and limitations of testing are discussed. This section concludes with a discussion of the PIXIT in a formal context.

2.2.1 The Meaning of Conformance

The starting point for conformance testing is the definition of what constitutes conformance. ISO9646 states that a conforming implementation is an implementation 'which satisfies both static and dynamic conformance requirements, ...' [ISO91a, part 1, sections 3.4.10 and 5.6] (see section 1.3.2).

It follows that

- the relevance of a protocol standard with respect to testing is its set of *conformance requirements*;
- a conforming implementation shall *satisfy* all conformance requirements.

In order to make this more formal, we will introduce some notation:

- A specification S in a particular standard can be written as

  $$S = \{r_1, r_2, r_3, \ldots\}$$

  where each r_i is a conformance requirement.
- The fact that an implementation I satisfies a particular conformance requirement r is denoted by

  $$I \ \textbf{sat} \ r$$

- A conforming implementation is one which satisfies all requirements in the standard:

  $$\forall r \in S : I \ \textbf{sat} \ r \tag{2.1}$$

  which will be abbreviated to

  $$I \ \textbf{sat} \ S$$

In a natural language specification the requirements are expressed in natural language, e.g.

$r =$ 'after receiving a Connect-Request-PDU the protocol shall respond with either a Connect-Indication-PDU or a Disconnect-PDU'

Formal requirements are expressed in a formal language with precisely defined syntax and semantics. Requirement r could be expressed in a fictitious language as

$$?\mathit{Con\text{-}Req\text{-}PDU} \rightarrow (!\mathit{Con\text{-}Ind\text{-}PDU} \ \vee \ !\mathit{Dis\text{-}Con\text{-}PDU})$$

where ? denotes receiving, ! sending, $\rightarrow$ sequentiality of actions, and $\vee$ choice between actions.

The formal language in which requirements are expressed is denoted by $\mathcal{L}_R$. Identifying a formal language with the set of all possible expressions in that language, so that $\mathcal{L}_R$ is the set of all possible requirements, we have that any specification S is a subset of $\mathcal{L}_R$:

$$S \subseteq \mathcal{L}_R$$

Languages that express requirements, or properties, are *logical languages*. Such languages allow expressions that are either *true* or *false* for a given system. Examples of logical languages that are used for protocol specification are Z [Spi89], Temporal Logic [Pnu86], and Hennessy-Milner Logic [HM85, Lar90]. Logical languages are powerful, however, in the design and specification of systems they have a disadvantage: they are non-constructive. This lack of constructivity makes it difficult to relate the structure of the specification to the structure of the system being specified, and consequently, implementation or prototyping based on such logical specifications is difficult.

This disadvantage is one of the reasons that current standardized FDTs (Estelle, LOTOS, SDL) are not based on logical languages. Formal descriptions in these FDTs do not define conformance requirements or properties; they define *observable behaviour*. For conformance testing, however, it is important to know which requirements are implicitly defined by the expressions in the FDT.

Let $\mathcal{L}_{FDT}$ be the language of the FDT, and let $B \in \mathcal{L}_{FDT}$ be an expression defining observable behaviour, then the fact that B specifies a particular requirement $r \in \mathcal{L}_R$ is written as

$$B \ \ \mathbf{spec} \ \ r$$

By introducing the relation **spec**, the set of all requirements specified by an expression $B \in \mathcal{L}_{FDT}$ is implicitly defined as:

$$S_B = \{r \in \mathcal{L}_R \mid B \ \ \mathbf{spec} \ \ r\} \tag{2.2}$$

Combining (2.1) and (2.2), we have for an implementation I that conforms to B:

$$\forall r \in \{r \in \mathcal{L}_R \mid B \ \ \mathbf{spec} \ \ r\} : I \ \ \mathbf{sat} \ \ r$$

or, equivalently:

$$\forall r \in \mathcal{L}_R : \ \ B \ \ \mathbf{spec} \ \ r \ \text{ implies } \ I \ \ \mathbf{sat} \ \ r \tag{2.3}$$

In this way conformance is defined as a relation between implementations and behaviour specifications, saying that an implementation conforms to a specification if every requirement specified by the behaviour specification is satisfied by the implementation.

Behaviour specifications are elements of the formal language $\mathcal{L}_{FDT}$. Implementations, on the other hand, are not necessarily formal objects. They can be non-formal realizations or concrete products. For the moment the only assumption that we make about implementations is the existence of a class of all possible implementations. Let *IMPL* be this class, then conformance can be expressed as a relation between *IMPL* and $\mathcal{L}_{FDT}$:

$$\textbf{conforms-to} \subseteq \mathit{IMPL} \times \mathcal{L}_{FDT} \tag{2.4}$$

defined by equation (2.3)

$$I \ \textbf{conforms-to} \ B \quad =_{def} \quad \forall r \in \mathcal{L}_R : \ B \ \textbf{spec} \ r \ \text{implies} \ I \ \textbf{sat} \ r \tag{2.5}$$

Given a class of implementations *IMPL* and a behaviour specification formalism $\mathcal{L}_{FDT}$ this still leaves freedom in defining conformance by varying $\mathcal{L}_R$, **spec**, and **sat**.

Note that $\mathcal{L}_R$ and $\mathcal{L}_{FDT}$ are different kinds of languages which are both necessary in this approach: to define conformance a logical language to express conformance requirements is needed; to specify behaviour in a constructive manner a behavioural specification formalism is needed.

Conformance in a Logical Model

A specification S is a set of requirements, i.e. a set of formulae in the logical language $\mathcal{L}_R$. This corresponds to the logical concept of a *theory* [Dal80]: given a logical language a theory is a set of formulae in that language.

The meaning of a logical language is usually given by a *satisfaction relation*, denoted by '$\models$'. This is a relation between a class of structures, the models of the language, and the formulae of the language, expressing the fact that a formula holds for a particular structure. In our case, the class of implementations *IMPL* may be considered to be the class of models, where the satisfaction relation is given by **sat**. With this interpretation a conforming implementation I is one that satisfies all formulae in S and it therefore corresponds to the logical concept of a *model* of S.

The same applies to **spec**: it could also be considered as a satisfaction relation '$\models$' when the class of models is taken to be $\mathcal{L}_{FDT}$. So it follows from (2.2) that also B, with the satisfaction relation **spec**, is a model of S.

The two satisfaction relations **sat** and **spec** are closely related but not identical: **spec** takes models from $\mathcal{L}_{FDT}$, which are formal objects, and **sat** takes models from *IMPL*, which could be non-formal realizations or concrete products.

Having S as a logical theory in the logical language $\mathcal{L}_R$ a lot of related concepts from logic can be introduced. An example is the definition of a *derivation system* for $\mathcal{L}_R$, i.e. a formal system of axioms and inference rules. It introduces the concept of *derivation* in $\mathcal{L}_R$:

$$S \vdash r$$

Preferably, the derivation system is defined in such a way that $S \vdash r$ if and only if every I that satisfies S also satisfies r, i.e. it should be *sound* and *complete*.

Other logical concepts, like the *consistency*, the *deductive closure*, and *logical independence* of a specification, are discussed in the examples in section 2.4.4.

2.2.2 PICS

As explained in section 1.3.2, most standards do not uniquely define the behaviour of a protocol, but they leave some space for an implementer to choose between different functions and options. The requirements on a minimum set of capabilities, and on consistency of the options chosen, are stated in the static conformance requirements of the standard. The PICS proforma lists all possibilities for the selection of options. The implementer states the implemented options in the Protocol Implementation Conformance Statement (PICS).

The allowance for different capabilities and options in standards means that specifications in standards are *parameterized*. Standards define a set of possible protocols, one possible protocol for each choice of options, i.e. for each choice of values for the parameters.

Comparing with ISO9646 terminology: the *PICS-proforma* is the *formal parameter* of the specification; the *PICS* is the *actual parameter*, which is substituted for the formal parameter for a particular protocol implementation. *Static conformance requirements* (*SCR*) define constraints on the value of the PICS.

Thus, we should write a protocol specification S as a parameterized specification, with formal parameter *PICS-proforma* of the *type*, i.e. the set of all possible correct values, determined by the static conformance requirements: *SCR-type*. Using a notation well-known from programming languages, this is expressed as

$$S(\mathit{PICS\text{-}proforma} : \mathit{SCR\text{-}type})$$

An instantiation of S for a particular implementation I with associated $PICS_I$ is given by

$$S(PICS_I)$$

The static conformance review corresponds to checking whether the *PICS* has a valid value, i.e. whether the *PICS* is of the type defined by the *SCR*: $\mathit{PICS} \in \mathit{SCR\text{-}type}$. This is analogous to parameter 'type-checking' in conventional programming languages.

The set of requirements $S_B(PICS_I)$ derived from the behaviour B defines the *dynamic* conformance requirements for implementation I.

This notion of parameterization straightforwardly extends to expressions in $\mathcal{L}_{FDT}$ defining behaviour:

$$B(\mathit{PICS\text{-}proforma} : \mathit{SCR\text{-}type})$$

Thus, a parameterized behaviour expression in $\mathcal{L}_{FDT}$ defines a function from *SCR-type* to behaviour specifications. Each correct instantiation defines a behaviour.

Summarizing, using formula (2.3): given a behaviour specification of a protocol in an FDT, $B(\textit{PICS-proforma} : \textit{SCR-type})$, and an implementation I with associated $PICS_I$, then

$$\begin{array}{rl} & I \text{ with } PICS_I \text{ conforms to } B(\textit{PICS-proforma} : \textit{SCR-type}) \\ =_{def} & \\ & PICS_I \in \textit{SCR-type} \\ \text{and} & \forall r \in \mathcal{L}_R : \;\; B(PICS_I) \;\; \textbf{spec} \;\; r \;\; \text{implies} \;\; I \;\; \textbf{sat} \;\; r \end{array} \tag{2.6}$$

2.2.3 Test Purposes

The first step towards the generation of an abstract test suite is the development of one or more test purposes for each conformance requirement (section 1.3.3).

Natural Language Test Purposes

ISO9646 introduces a test purpose as 'a prose description of a narrowly defined objective of testing, focusing on a single conformance requirement.' [ISO91a, part 1, section 3.6.5]. To analyse how test purposes are obtained from a conformance requirement we will study an example.

First observe that in a natural language specification all conformance requirements are mentioned explicitly in the standard, and thus, S is necessarily finite. Suppose that a standard mentions as such a requirement

'7-bit data-packets shall be sent in 8-bit PDUs,
where the 8^{th} bit is the parity bit' (2.7)

From this requirement many additional requirements can be derived. In this case we can derive 128 requirements:

0000 000 shall be sent as 0000 0000
0000 001 shall be sent as 0000 0011
⋮

The original requirement (2.7) is untestable in practice. For testing we do not take (2.7), but a selection of the derived requirements. Three test purposes selected from the derived requirements of (2.7) are:

test whether 0000 000 is sent as 0000 0000
test whether 1111 111 is sent as 1111 1111
test whether 1001 100 is sent as 1001 1001

Thus, in the derivation of test purposes two steps can be recognized:

- derivation of (additional) testable requirements
- selection of testable requirements

Formal Test Purposes

Formally, the set of dynamic conformance requirements S_B of a behaviour specification B for an implementation with associated $PICS_I$, is given by equation (2.2):

$$S_B \;=\; \{r \in \mathcal{L}_R \mid B(PICS_I)\ \ \mathbf{spec}\ \ r\}$$

First observe that contrary to a natural language specification S_B need not, and in general will not, be finite. Normally there will be infinitely many formulae that B specifies (see also the examples in section 2.4).

In our interpretation of **spec**, S_B includes all requirements expressible in the language $\mathcal{L}_R$ that are specified by B, whether this occurs directly or indirectly. Therefore additional requirements cannot be derived from S_B, so the first step in the derivation of test purposes as given for the natural language method does not exist here. Thus, deriving test purposes amounts to selecting a 'testable' subset P of S_B. This subset P should be finite, since for each test purpose at least one test case will be generated, and we would like to end up with a finite set of test cases:

$$P \subseteq S_B, \quad P = \{p_1, p_2, \ldots, p_m\}$$

The selection of a testable set of test purposes $P \subseteq S_B$ is crucial, e.g. for the coverage of the generated abstract test suite. On the other hand this aspect is the most difficult to formalize. Heuristic methods will have to guide the selection process. [ISO91a, part 2, section 10.4] gives hints on the selection of test purposes. The formalization of the selection process is studied in chapter 6.

2.2.4 Test Cases

Once a finite set of test purposes $P = \{p_1, p_2, \ldots, p_m\}$ that adequately covers all aspects of the protocol has been selected, the next step is to determine whether I **sat** p_i for each test purpose p_i. In principle, there are two ways to determine this, viz. by *formal verification* and by experimentation or *testing* (cf. section 1.1.2). Formal verification means that mathematical methods are used to determine whether I **sat** p_i. A requisite for using verification is that the implementation I is a formal object. Since our implementations are mostly non-formal objects, which are not amenable to formal verification, we will have to rely on testing. This means that for each test purpose p_i an experiment or test t_i has to be found such that from the observations of testing I with t_i satisfaction of p_i can be concluded.

Test Application

A test case t specifies behaviour to be performed by an environment of the implementation, hence it is expressed in some behavioural formalism $\mathcal{L}_T$, which is referred to as the *test notation*. $\mathcal{L}_T$ could be (but need not necessarily be) the same as $\mathcal{L}_{FDT}$. It should be powerful enough to describe any environment of the implementation and to specify experiments performed by that environment.

Let $t \in \mathcal{L}_T$ be a test case, and let I be the implementation under test, then we denote the *application* of t to I by

$$apply(t, I)$$

The result of $apply(t, I)$ is a *verdict*, i.e. a statement about the success or failure of the application of the test case to I. The function *apply* models the execution of a test case.

Test Validity

Developing test cases means that for each test purpose $p_i \in P$, $1 \leq i \leq m$, we have to find a test case $t_i \in \mathcal{L}_T$ such that from the observations of the application of t_i to I satisfaction of p_i can be concluded, i.e. we have to find a test case t_i that is *valid* with respect to p_i. Validity means that a test case really tests what it is intended for.

If, analogous to ISO9646, successful application is given the verdict **pass**, and unsuccessful application the verdict **fail**, then validity of a test case t with respect to a test purpose p is expressed by:

$$apply(t, I) = \textbf{pass} \quad \text{iff} \quad I \ \textbf{sat} \ p \tag{2.8}$$

This kind of validity is called *strong validity*. It assumes that the result of test application is either **pass** or **fail**, i.e. that satisfaction of the test purpose can always be decided from application of the corresponding test case. In some situations it may happen that this decision can not be made based on testing. Due to limitations in observability and controllability it is possible that no evidence of failure can be found, while on the other hand also satisfaction of the test purpose can not be concluded. For such cases we allow a 'don't know' result of $apply(t, I)$. Analogous to ISO9646 it is expressed by the verdict **inconclusive**.

A consequence is that the right-to-left implication of equation (2.8) does not hold: if I **sat** p then $apply(t, I)$ can be **pass** or **inconclusive**. We adapt equation (2.8) by saying that a test case is valid if it is not the case that I **sat** p and $apply(t, I) = \textbf{fail}$, or I **sat** p and $apply(t, I) = \textbf{pass}$. We refer to this notion of validity as *weak validity*:

$$\begin{cases} apply(t, I) = \textbf{fail} & \text{implies} \quad I \ \textbf{sat} \ p, \quad \text{and} \\ apply(t, I) = \textbf{pass} & \text{implies} \quad I \ \textbf{sat} \ p \end{cases} \tag{2.9}$$

Test Validation and Test Hypothesis

The process of checking the validity of test cases with respect to their test purposes is called *test validation*. For test validation we have to verify equation (2.8), respectively (2.9), for all possible implementations. However, implementations as they occur in equations (2.8, 2.9) can be non-formal objects, not amenable to verification. It is only possible to verify these equations for all possible implementations, if we put an additional assumption on our class of implementations *IMPL*: we assume that we can model implementations by a formalism $\mathcal{L}_{IMPL}$ with which we can represent implementations, and which is amenable to verification. Let each possible implementation I be modelled by an element $B_I \in \mathcal{L}_{IMPL}$, such that all relevant aspects of I are represented by B_I, then checking a test case t for strong validity with respect to a test purpose p corresponds to verifying

$$\forall B_I \in \mathcal{L}_{IMPL}: \quad apply(t, B_I) = \mathbf{pass} \qquad \text{iff} \qquad B_I \ \mathbf{sat} \ p \tag{2.10}$$

Thus the function $apply : \mathcal{L}_T \times \mathcal{L}_{IMPL} \rightarrow \{\mathbf{pass}, \mathbf{fail}\}$ models the application of test cases to *models* of implementations. Analogously **sat** can now be stated more precisely as a relation between models of implementations and requirements: $\mathbf{sat} \subseteq \mathcal{L}_{IMPL} \times \mathcal{L}_R$.

The analogous requirement for weak validity is:

$$\forall B_I \in \mathcal{L}_{IMPL}: \quad \begin{cases} apply(t, B_I) = \mathbf{fail} & \text{implies} \quad B_I \ \mathbf{sat} \ p, \quad \text{and} \\ apply(t, B_I) = \mathbf{pass} & \text{implies} \quad B_I \ \mathbf{sat} \ p \end{cases} \tag{2.11}$$

where $apply : \mathcal{L}_T \times \mathcal{L}_{IMPL} \rightarrow \{\mathbf{pass}, \mathbf{fail}, \mathbf{inconclusive}\}$.

If we are looking for a language $\mathcal{L}_{IMPL}$, the first candidate is $\mathcal{L}_{FDT}$. The language $\mathcal{L}_{FDT}$ is used to specify unambiguously the relevant, external behaviour of protocols. As such it can also be used to describe the behaviour of implementations of these protocols. Other choices for $\mathcal{L}_{IMPL}$ are possible. An example is a specification that contains nondeterministic behaviour, whereas implementations are assumed to behave completely deterministically. In such a case a language can be chosen for $\mathcal{L}_{IMPL}$ that does not allow nondeterministic behaviour, e.g. a suitable sublanguage of $\mathcal{L}_{FDT}$.

The assumption that any concrete I can be represented by some model in $\mathcal{L}_{IMPL}$ is referred to as the *test hypothesis* (cf. [Ber91]). Intuitively, it is the assumption that we know something about our implementation under test, in this case that it is an implementation of a protocol, and not something completely different, and that such implementations can be conveniently modelled using $\mathcal{L}_{IMPL}$. We can also say that the test hypothesis expresses that the IUT is 'sufficiently close to the specification' to make testing useful [Ber91]. Without such a test hypothesis functional, black box testing, i.e. testing based on specifications, is generally impossible.

An example of a test hypothesis is that we assume our implementation to behave like a finite state machine (FSM), in case we have an FSM specification. We do not know which FSM is a model of the implementation, only that there is such a model, which we could construct if we had enough knowledge about the interior of the implementation.

Some algorithms for test generation from FSMs make even stronger assumptions, e.g. on the number of states in the FSM modelling the implementation. It is assumed that this number is less than m times the number of states in the specification, where m must be known [Cho78]. The larger m is chosen, the weaker the test hypothesis is, the more possible implementations are considered, and the larger the number of test cases is that the algorithm generates. The test hypothesis serves as a *test selection criterion*.

Test Runs

The *test application* of a test case t to an implementation I as modelled by the function *apply* may consist of executing several *test runs*. A test run is understood as one execution of the test case with the implementation under test. Due to nondeterminism a particular test run may lead to behaviour which is correct but which is not intended in the sense that no observation can be made from which satisfaction of the test purpose can be concluded. An observation of a test run can be anything, e.g. **pass**, **fail**, or **inconclusive**, logs of all occurred events, the occurrence of deadlock, etc. Test application of a test case consists of one or more test runs with the same test case, and assignment of the verdict **pass** or **fail**, based on the combined observations of all test runs.

To express this formally we first introduce a class Ω of possible observations of test runs. Since different test runs may lead to different observations in Ω, a test run is a relation $run \subseteq \mathcal{L}_T \times IMPL \times \Omega$, with $run(t, I, \omega)$ expressing the fact that a test run of test case t with implementation under test I may lead to observation ω. All possible observations of t and I are collected in a set

$$runs(t, I) = \{\ \omega \in \Omega \mid run(t, I, \omega)\ \}$$

The set of results of all runs with the same test case is evaluated, culminating in a verdict for test application:

$$apply(t, I) = eval(runs(t, I)) \tag{2.12}$$

An example of such an evaluation when each test run can have as a result **pass**, **fail**, or **inconclusive**, is:

$$eval(V) = \begin{cases} \textbf{fail} & \text{if } \textbf{fail} \in V \\ \textbf{pass} & \text{if } \textbf{fail} \notin V \end{cases}$$

See also the examples in section 2.4.5.

2.2.5 Generic Test Cases

In the test generation process ISO9646 distinguishes between different kinds of test cases. As a first step towards finding a set of abstract test cases, i.e. a test suite that is implementation independent and suitable for standardization, ISO9646 recommends

that a generic test case[1] be developed for each test purpose (section 1.3.3). The set of generic test cases is the *generic test suite GTS*:

$$GTS = \{g_1, g_2, \ldots, g_m\}$$

A *generic test case* g_i is the specification of an experiment that could be applied to I with ideal controllability and observability. In fact, the test case g_i is the operational 'mirror-image' of the test purpose p_i: whereas p_i describes the required property of I, g_i describes the required behaviour of the environment of I in order to test for p_i (figure 2.1).

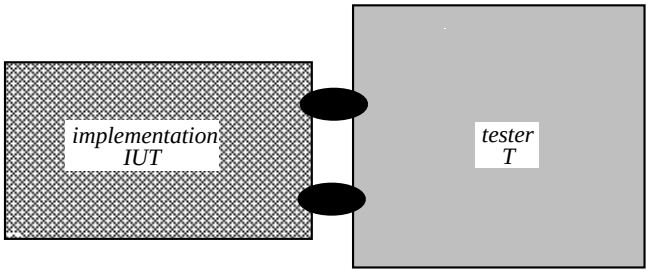


Figure 2.1: Testing with generic test cases.

Developing a generic test suite means that for each test purpose $p_i \in P$, $1 \leq i \leq m$, we have to find a generic test case $g_i \in \mathcal{L}_T$ that is valid with respect to p_i.

Validity of Generic Test Cases

A generic test case g_i specifies behaviour to be performed by an ideal environment of the implementation. Since conformance requirements, and thus test purposes, should only specify the external behaviour of the implementation, i.e. how the implementation interacts with its environment, it follows that a generic test case must always be able to decide about satisfaction of the corresponding test purpose. Saying it the other way around: if there is no generic test case, i.e. no ideal environment that can distinguish between implementations satisfying and not satisfying a test purpose, then there is no environment at all that can make the distinction. Hence such a test purpose does not specify a requirement on the external behaviour of the implementation.

As a consequence the 'don't know' verdict **inconclusive** is not appropriate as a result of the application of a generic test case; the result is always **pass** or **fail**. Hence for generic test cases strong validity is required (2.10)[2].

[1] [ISO91b] introduces the term *conceptual test case* for this kind of 'ideal' test case, in order to avoid confusion with [ISO91a], where in some contexts 'generic' is used with a slightly different meaning. 'Conceptual' refers to the conceptual testing architecture in [ISO91a, part 1, section 7.3.1].

[2] This notion of validity of generic test cases differs from [TKB92] (which agrees with [ISO91b]), where weak validity is required. Following the above argumentation we are in favour of requiring strong validity for generic test cases.

Being an ideal environment means that a generic test case is not necessarily realizable in practice. Consequently, application of a generic test case may correspond to *virtual application*: e.g. it may include infinitely many runs of a test case with the implementation, or runs of infinite length.

2.2.6 Abstract Test Cases

The final step in the test generation process (section 1.3.3) is the derivation of an abstract test case for each generic test case. This involves the transformation of each generic test case g_i into an abstract test case a_i, taking into account the testing environment in which an implementation is tested, i.e. taking into account the test method.

As explained in section 1.3.3 different aspects can be distinguished in the test method: accessibility of the boundaries of the IUT, the service boundaries from where the IUT can be observed and controlled, positioning of the tester in the SUT, and the distribution of testing functions. These aspects restrict the way in which the tester can control and observe the implementation:

1. When the tester is positioned in another computer system than the IUT, testing must be done via an underlying service provider. When the service boundaries from where the IUT can be observed and controlled do not coincide with the boundaries of the IUT, testing must be done via the surrounding protocol layers of the IUT. In both cases the access points of the IUT are not directly accessible to the tester. There is something between the tester and the IUT: the *test interface* or *test context* (figure 2.2, cf. figure 2.1).
2. Accessibility of the boundaries of the IUT can be restricted, in the sense that some access points of the implementation are not accessible at all. An example is the upper service boundary in the remote test method.
3. Distribution of access points at the boundary of the implementation imposes a distribution of testing functions. An example is the distribution between an upper SAP (service access point) and a lower SAP. A test coordination procedure is needed to coordinate these testing functions, e.g. between an upper tester and a lower tester. This distribution has some analogy with the decomposition of a service into protocol entities. It may cause difficulties in relating observations made by the distributed testers, e.g. the order of two events cannot be reconstructed if they are observed by separate testers [BDZ89].

Let the implementation I have the access points (e.g. SAPs) $iap_1, iap_2, \ldots, iap_k$ on its boundary. They make up the boundary as given by the specification. We call them *Implementation Access Points*, and the complete set of them:

$$IAPs = \{iap_1, iap_2, \ldots, iap_k\}$$

Because of restriction (2) we can only access a subset of them:

$$IAPs' \subseteq IAPs$$

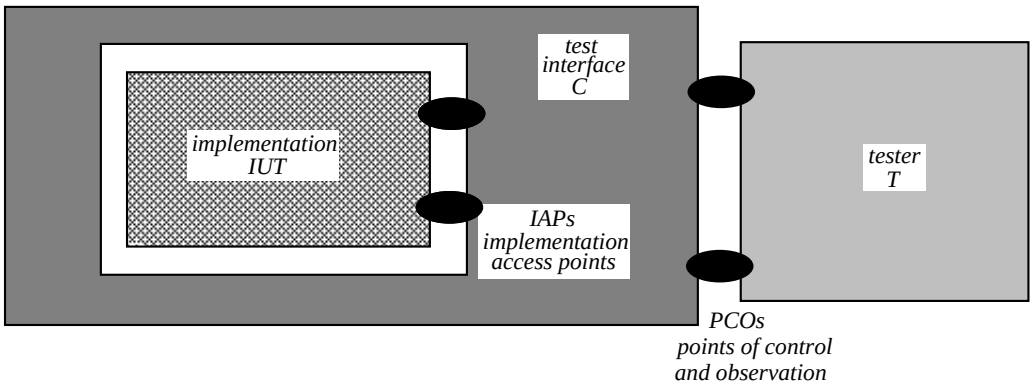


Figure 2.2: Test interface.

The $IAPs'$ cannot be accessed directly, but, because of restriction (1), only via the test interface. The points on the boundary of the test interface that are accessible to the tester are the *Points of Control and Observation* ([ISO91a, part 1, section 3.8.1], section 1.3.3):

$$PCOs = \{pco_1, pco_2, \ldots, pco_l\}$$

The test interface C relates the actions at the $PCOs$ and the actions at the $IAPs'$. In order to base the transformation from generic to abstract test cases on formal methods, a formal description of the behaviour of C is needed, either as an expression $B_C \in \mathcal{L}_{FDT}$, or as a set of properties $S_C \subseteq \mathcal{L}_R$. Conformance of the implementation of the test interface to this description is normally only assumed, and not checked. This assumption is part of the test hypothesis (section 2.2.4) for abstract testing.

Restriction (3) leads to a partition of $PCOs$ (a division of $PCOs$ in disjoint subsets, together forming the whole set $PCOs$), where each subset represents a (geographical) location.

Example 2.1
Figure 2.3 shows an instance of this general abstract testing architecture. The implementation I interacts with its direct environment, the test interface C, via three implementation access points: IAP_1, IAP_2, and IAP_3. There are two points on the boundary of the test interface that are accessible to the tester: PCO_1 and PCO_2. Only the subset $\{IAP_1, IAP_2\} \subseteq \{IAP_1, IAP_2, IAP_3\}$ can be accessed via $\{PCO_1, PCO_2\}$: IAP_1 can be controlled and observed through PCO_1, IAP_2 can be controlled and observed through PCO_2, and IAP_3 cannot be controlled or observed by the tester. The testing functions are distributed over T_1 and T_2, with a test coordination procedure to establish the combined operation of T_1 and T_2 as observed by the test interface.

In figure 2.4 two concrete instances of the testing architecture are shown. Figure 2.4(a) shows the remote test method [ISO91a, part 2, section 12.3.5]. In the remote test method the IUT is tested via the underlying service. There is only one PCO: the remote SAP of the underlying service. It controls and observes IAP_1, the lower SAP of

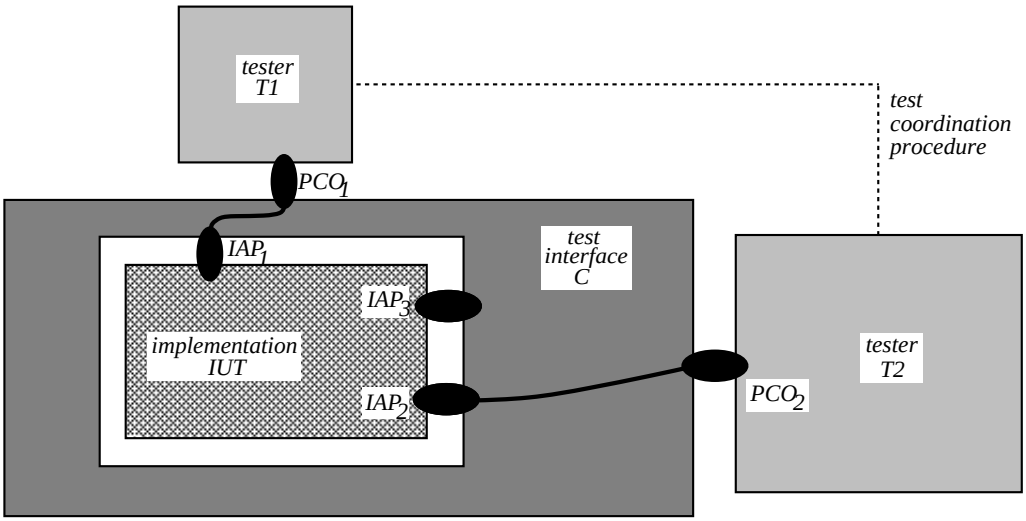


Figure 2.3: Test interface with distributed testers.

the IUT, via the underlying service provider, which constitutes the test interface. The remote test method does not require access to the upper service boundary of the IUT; IAP_2, the upper SAP of the IUT, is not accessible to the tester.

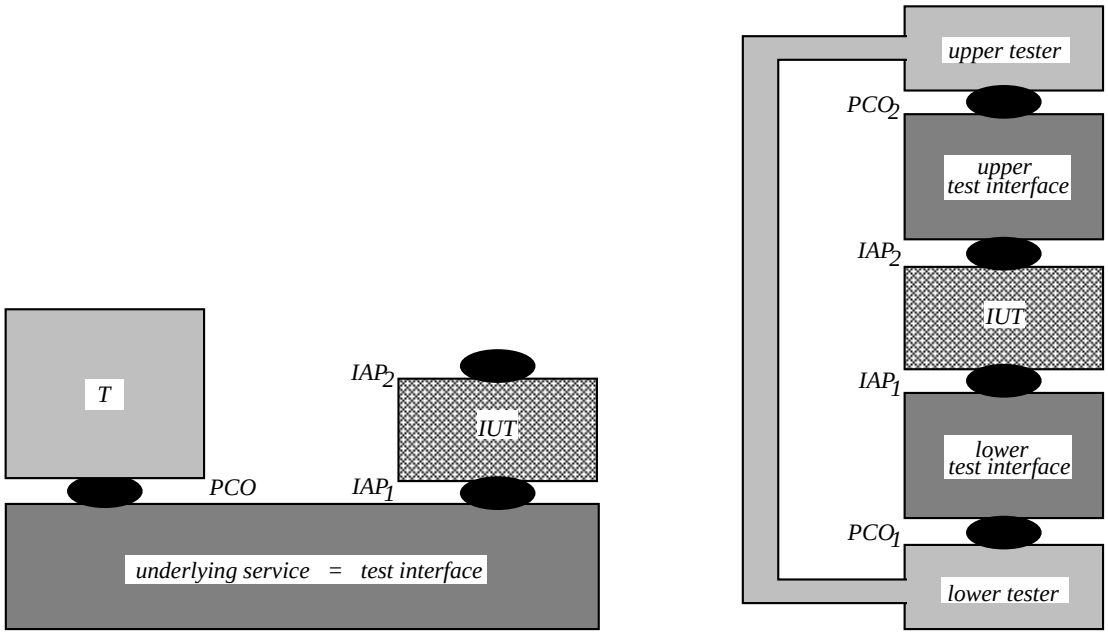
In figure 2.4(b) both PCOs are situated in the SUT, while the IUT is embedded in other protocol layers. The test interface consists of two parts: the protocols between the upper SAP of the IUT and the upper tester constitute the upper part of the test interface, and the protocols between the lower SAP and the lower tester constitute the lower part of the test interface. The test coordination procedure is realized entirely within the test system.

□

Developing abstract test cases means that for each generic test case $g_i \in GTS$, $1 \leq i \leq m$, we have to find an abstract test case a_i, such that the result of applying a_i is valid with respect to the (virtual) application of g_i. Generic test cases are generally too ideal for a realistic testing environment, where controllability and observability of the IUT are limited. Whereas a generic test case g_i is expressed in terms of control and observation of *IAPs*, the corresponding abstract test case a_i is expressed in terms of control and observation of *PCOs*. One could also say that for generic test cases *IAPs* and *PCOs* coincide.

Validity of Abstract Test Cases

Because of the limitations in controllability and observability it is possible that using an abstract test case we cannot conclude about satisfaction of a test purpose, while using the corresponding generic test case we can. Representing the execution of an abstract test case $a \in \mathcal{L}_T$ by the function *apply* (section 2.2.4), this implies that **inconclusive**



(a) Remote test method *(b) Embedded testing*

Figure 2.4:

can occur as verdict, and validity is defined by (2.11). Hence, the resulting verdict of applying an abstract test case a need not be the same as the verdict of its corresponding generic test case g, but it may not contradict the result of the generic test case:

$$\forall B_I \in \mathcal{L}_{IMPL} : \begin{cases} apply(a, B_I) = \textbf{fail} & \text{implies} \quad apply(g, B_I) = \textbf{fail}, \quad \text{and} \\ apply(a, B_I) = \textbf{pass} & \text{implies} \quad apply(g, B_I) = \textbf{pass} \end{cases} \tag{2.13}$$

Note that introducing **inconclusive** as a verdict allows that an abstract test case that always gives the verdict **inconclusive** is valid. However, such a test case is not useful, in the sense that it will never extract any information about the IUT that is being tested. In deriving abstract test cases we are not only interested in valid test cases, but also in useful test cases, i.e. test cases that extract as much information as possible from the IUT given the limitations of the test method.

Remark 2.2
Due to limitations in observability and controllability introduced by a test method, it may even happen that no useful abstract test case can be specified for a generic test case g. This means that application of g to the implementation I, $apply(g, I)$, cannot be realized in practice for that test method.

For the same reason, the second requirement for a to be valid with respect to a test purpose p (2.11), viz.

$$apply(a, I) = \textbf{pass} \quad \text{implies} \quad I \ \textbf{sat} \ p$$

may be too strong. Making a definite decision I **sat** p based on testing may sometimes be practically unfeasible. In such a case we leave the mathematical rigour of equation (2.11), and assign the verdict **pass** when we have enough confidence that I **sat** p. By elaborating these thoughts we enter the fields of *probabilistic testing*: application of a test case gives a probability that the implementation has a certain property, and of *probabilistic specification*: a requirement specifies the probability with which an implementation shall have a certain property [LS89, Chr90a].

□

2.2.7 Test Generation in Practice

In figure 2.5 the test suite generation process as proposed in the previous sections is sketched. This process is a kind of 'normal form' (this term comes from [ISO91a, part 2, section 8]) for test suite generation from formal specifications. It provides a formal basis for the generation of valid test cases. However, generation of test cases for a particular protocol specified by $B(\textit{PICS-proforma} : \textit{SCR-type})$ need not strictly follow this strategy, in which all intermediate steps are produced explicitly. This may not even be possible since the number of requirements in S_B could be infinite. The given strategy must be seen as a guide-line and a reference point for correctness of a given algorithm for test case generation. In case of formal methods it is necessary that any algorithm used will result in *provably* correct results, referring to the normal form for the test generation process.

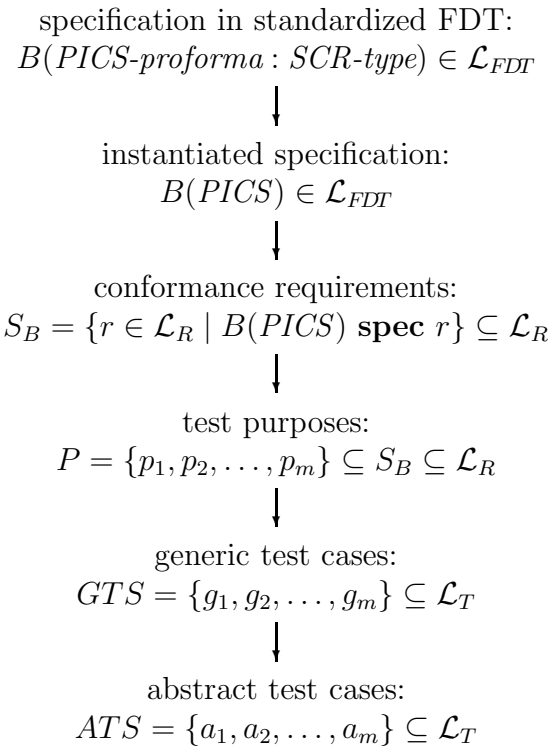
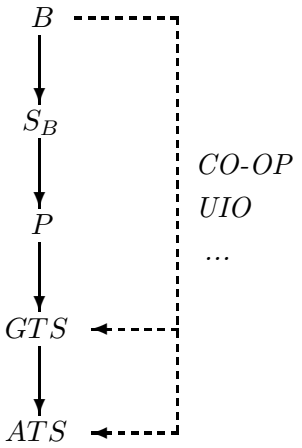
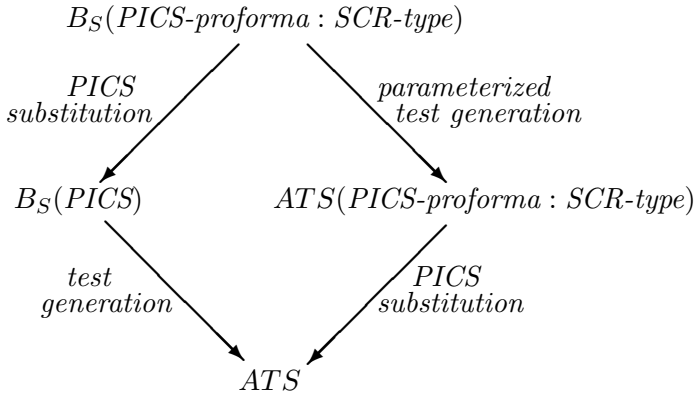


Figure 2.5: Overview of the 'normal form' of the test generation process.

Examples of test generation techniques that generate test cases directly from the behaviour specification $B \in \mathcal{L}_{FDT}$, without first deriving conformance requirements and test purposes explicitly, are Unique Input/Output sequences for finite state machines [ADLU88, BU91], the CO-OP method for LOTOS [Wez90], and the methods for test generation that will be presented in the chapters 4 and 5. Such methods can be used if they can be proven to give results which are equivalent to the ones obtained by the normal form test generation process (figure 2.6(a)).



(a) Relation with existing test generation methods



(b) Substitution of PICS

Figure 2.6:

Substitution of PICS

Until now it was assumed that the *PICS* of an IUT is given before test generation could start. If we allow parameterized or *open* requirements in $\mathcal{L}_R$ we can also try to perform parameterized test generation, postponing substitution of *PICS* for *PICS-proforma*. The result will be a parameterized abstract test suite[3]

$$ATS(PICS\text{-}proforma : SCR\text{-}type)$$

This way of test generation is proposed in ISO9646, thus avoiding redoing test generation for each different value of the *PICS*, which is important since the tests are generated manually. Once algorithms for test generation can be defined, and tools that support these algorithms are available, it does not matter any more when the *PICS* is

[3]'Parameterized' means that there is a formal parameter, contrary to the use of 'parameterized' in ISO9646 (for example in 'parameterized abstract test suite'), where it means that 'a parameter value is supplied in accordance with the PICS/PIXIT'. The usual term for this value supply is 'instantiation' or 'application'.

substituted as long as the result is the same, i.e. the diagram in figure 2.6(b) should commute.

2.2.8 Limitations of Testing

If we succeed in deriving a generic test suite GTS that has a test case for each of the test purposes in the selected set $P \subseteq S$, and an abstract test suite ATS that is valid with respect to GTS, then we may conclude that

I conforms to S	iff	$(*$ by equation (2.1) $*)$
$\forall r \in S : I$ **sat** r	implies	$(*$ since $P \subseteq S$ $*)$
$\forall p \in P : I$ **sat** p	iff	$(*$ by equation (2.10) $*)$
$\forall g \in GTS : apply(g, I) \neq$ **fail**	implies	$(*$ by contraposition of (2.13) $*)$
$\forall a \in ATS : apply(a, I) \neq$ **fail**		

Therefore by contraposition:

$$\exists\, a \in ATS : apply(a, I) = \textbf{fail} \quad \text{implies} \quad I \text{ does not conform to } S$$

The fact that the implication is valid only in one direction ('if we have a test case with **fail**-result then we have a non-conforming implementation', but not 'if we have a non-conforming implementation then we have a test case with **fail**-result') reflects the famous dictum that 'testing can only show the presence of errors, not their absence'.

2.2.9 PIXIT

The PIXIT contains extra information about the IUT and its testing environment, needed in the test implementation phase (section 1.3.4). ISO9646 states about the PIXIT: 'A statement made by a supplier or implementer of an IUT which contains or references all of the information (in addition to that given in the PICS) related to the IUT and its testing environment, which will enable the test laboratory to run an appropriate test suite against the IUT' [ISO91a, part 1, section 3.4.8]. We can also describe the PIXIT as all parameters of the executable test suite that are not parameters of the specification (since specification parameters constitute the PICS).

Two kinds of parameters can be distinguished:

1. Parameters that describe the relation between abstract concepts in the abstract test cases and concrete concepts in the implementation and its testing environment. Examples are the mapping of *PCOs* to concrete addresses, memory locations, or file descriptors, and the mapping of ASPs (abstract service primitives) to concrete data structures.

   In a formal context this corresponds to giving the relation between the formal objects of the specification formalism (signals, gates, abstract data types, ...) and the concrete objects of the implementation. This can be seen as an *interpretation function* from formal objects to concrete objects.

2. Parameters that add precision to very general requirements in the specification for the purpose of testing. Such requirements can be of the form 'there must be an x such that $P(x)$'. To test $P(x)$ it can be necessary to know which value of x has been chosen in the implementation, especially if the value domain of x is very large. Examples are the largest representable integer, the exact value of which can be important for a particular test case, and the requirement 'the implementation shall support N connections, where N shall be at least 3'; in the PIXIT the exact value for N is given in order to do meaningful testing.

   In a formal context this corresponds to the occurrence of *nondeterminism* in specifications. Sometimes the nondeterministic specifications leave open too many choices to do meaningful testing. The PIXIT defines a *restriction of nondeterminism*, indicating how the nondeterminism was resolved in a particular implementation.

2.3 Conformance as a Relation

In section 2.2.1, (2.4) and (2.5), it was shown that conformance can be considered as a relation **conforms-to** between the class of implementations $IMPL$ and $\mathcal{L}_{FDT}$. In section 2.2.4 we assumed that the behaviour of the 'real' implementation I can be represented by the formal expression $B_I \in \mathcal{L}_{IMPL}$:

$$\textbf{conforms-to} \subseteq \mathcal{L}_{IMPL} \times \mathcal{L}_{FDT}$$

Since a natural choice for $\mathcal{L}_{IMPL}$ is $\mathcal{L}_{FDT}$, **conforms-to** can be considered as a relation *on* $\mathcal{L}_{FDT}$:

$$\textbf{conforms-to} \subseteq \mathcal{L}_{FDT} \times \mathcal{L}_{FDT}$$

which according to equation (2.5) is defined by

$$B_I \textbf{ conforms-to } B_S \quad =_{def} \quad \forall r \in \mathcal{L}_R : \; B_S \textbf{ spec } r \;\text{ implies }\; B_I \textbf{ sat } r \qquad (2.14)$$

This definition depends on the requirement language $\mathcal{L}_R$ and the relations **spec** and **sat**. Different choices for these allow the definition of different classes of conforming implementations with the same language $\mathcal{L}_{FDT}$. This is illustrated in the examples in section 2.4.2.

Once $\mathcal{L}_R$, **spec**, and **sat** have been chosen the relation is fixed. We can try to find a characterization of this relation in terms of behaviour expressions only, so that conformance can be studied as a relation between behaviour descriptions, without considering requirements explicitly. The language $\mathcal{L}_R$ together with **spec** and **sat** only serves as the basis for defining the conformance relation; in studying conformance we can do without it. In studying behavioural formalisms this approach of considering conformance as a relation endowed with certain properties is common.

If **conforms-to** is a relation on $\mathcal{L}_{FDT}$, **spec** and **sat** are both relations between $\mathcal{L}_{FDT}$ and $\mathcal{L}_R$. This makes it possible to compare **spec** and **sat**, and to choose **spec** equal

to **sat**. The next proposition relates properties of **conforms-to** to properties of **spec**, **sat** and $\mathcal{L}_R$.

Proposition 2.3
Let **conforms-to** $\subseteq \mathcal{L}_{FDT} \times \mathcal{L}_{FDT}$ be defined by (2.14):

$$B_I \textbf{ conforms-to } B_S \quad =_{def} \quad \forall r \in \mathcal{L}_R : \ B_S \textbf{ spec } r \ \text{ implies } \ B_I \textbf{ sat } r$$

then

1. **conforms-to** is *reflexive* if and only if **spec** $\subseteq$ **sat**
2. **conforms-to** is *transitive* if **spec** $\supseteq$ **sat**
3. **conforms-to** is a *preorder* if **spec** $=$ **sat**
4. **conforms-to** is an *equivalence* if **spec** $=$ **sat** and negation is expressible in $\mathcal{L}_R$, i.e. if
$$\forall r \in \mathcal{L}_R, \exists\, \overline{r} \in \mathcal{L}_R, \forall B \in \mathcal{L}_{FDT} : B \textbf{ sat } \overline{r} \quad \text{iff} \quad \text{not} \ \ B \textbf{ sat } r$$

□

The converse of proposition 2.3.2 (and thus of 2.3.3) does not hold as follows from the counter example:

> **conforms-to** $= \mathcal{L}_{FDT} \times \mathcal{L}_{FDT}$, **spec** $= \emptyset$, and **sat** $\neq \emptyset$;
> then **conforms-to** is transitive, (2.14) is fulfilled and **spec** $\not\supseteq$ **sat**.

If the additional requirement is put that every set of requirements that is specified by a B_S, is exactly satisfied by a B_I, and vice versa, then the converse does hold:

Proposition 2.4
If $\forall B_I \in \mathcal{L}_{FDT}, \exists\, B_S \in \mathcal{L}_{FDT} : \{r \in \mathcal{L}_R \mid B_S \textbf{ spec } r\} = \{r \in \mathcal{L}_R \mid B_I \textbf{ sat } r\}$,
and $\forall B_S \in \mathcal{L}_{FDT}, \exists\, B_I \in \mathcal{L}_{FDT} : \{r \in \mathcal{L}_R \mid B_I \textbf{ sat } r\} = \{r \in \mathcal{L}_R \mid B_S \textbf{ spec } r\}$,
then

$$\textbf{conforms-to} \ \text{ is transitive} \quad \text{if and only if} \quad \textbf{spec} \supseteq \textbf{sat}$$

□

We see that the two approaches in the specification of concurrent systems, viz. the *logical approach* in which a specification is a set of formulae in some logic, and conformance is expressed by satisfaction of these formulae, and the *behavioural approach*, where specifications are expressions with an operational, behavioural interpretation, and conformance is based on a comparison between the observable behaviours of the specification and the implementation, formalized as a relation, are closely related [Lar90].

Whereas (the formal interpretation of) ISO9646 uses the logical approach, current FDTs are based on the behavioural approach. Conformance as a relation, and more specifically as a preorder (proposition 2.3.3) on the FDT, is often studied in literature [BAL$^+$90, Led90]. In [BAL$^+$90] it is referred to as an *implementation relation*, and denoted by $\leq_{\mathcal{R}}$.

2.4 Examples

In this section simple examples of vending machines are used to illustrate the ideas presented in the previous sections. The examples are based on the formalism of *labelled transition systems* (section 1.4), using the vending machine specified by the action tree of figure 2.7.

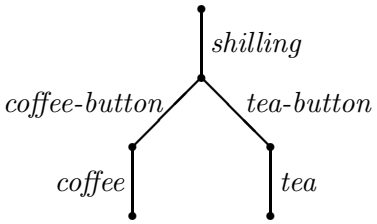


Figure 2.7: Action tree of a vending machine.

First, in section 2.4.1, a specification is given by means of natural language requirements, and the ISO9646 abstract test suite generation trajectory (section 1.3.3) is followed. In section 2.4.2 the labelled transition system of figure 2.7 is taken as its formal behaviour specification. This means that we choose $\mathcal{L}_{FDT}$ to be $\mathcal{LTS}$. Formal requirements are derived using different requirement languages $\mathcal{L}_R$. It is shown that different languages $\mathcal{L}_R$ allow different classes of conforming implementations of the vending machine in figure 2.7. Implementations are also represented as labelled transition systems. This means that $\mathcal{L}_{IMPL}$ is chosen to be $\mathcal{LTS}$, and that the test hypothesis is: $I \in \mathcal{LTS}$. Some of the classes of implementations are shown to correspond to relations defined in section 1.4.3. Variations of **spec** and **sat** will lead to even more, different classes of conforming implementations. Section 2.4.3 gives an example of introducing optional behaviour in the vending machine. In section 2.4.4 the logical concept of derivation is used to reduce the number of requirements specified by figure 2.7. Finally, in section 2.4.5 a generic and abstract test case are derived.

2.4.1 Natural Language Specification of the Vending Machine

In this subsection we will consider a natural language specification in English of the vending machine. Natural language conformance requirements are given, and from these a test purpose, a generic test case, and an abstract test case are derived according to the ISO9646 methodology.

Specification

Let the natural language specification of the vending machine M be given by the following conformance requirements:

r_1: an implementation of M must be able to accept a *shilling*;

r_2: after an implementation of M has accepted a *shilling* there is a choice between pushing the *coffee-button* and the *tea-button*;

r_3: after the *coffee-button* has been pushed the machine shall produce *coffee*;

r_4: after the *tea-button* has been pushed the machine shall produce *tea*.

Test Purposes

The first step in generating an abstract test suite from these requirements is the selection of test purposes (section 1.3.3). An example of a test purpose for requirement r_3 is

- test whether the machine produces *coffee* after the *coffee-button* has been pushed.

Generic Test Cases

For a corresponding generic test case we assume that the IUT is in a state where the *coffee-button* can be pushed. Then the test case is

1. push the *coffee-button*;
2. check whether *coffee* is produced.

Abstract Test Cases

To obtain an abstract test case we first have to expand the assumption of the generic test case: bringing the IUT in the desired state. This is the preamble of the abstract test case. It consists of the action *shilling*. The resulting abstract test case can be described in a TTCN-like notation ([ISO91a, part 3], section 1.3.3):

behaviour	constraints	verdict
!*shilling*		
!*coffee-button*		
?drink	drink = *coffee*	**pass**
?drink	drink $\neq$ *coffee*	**fail**

The first event is *shilling*, the exclamation mark indicating that the initiative for this event is with the tester. After the second event (*coffee-button*) a *drink* shall be received by the tester. There are two alternatives. If *drink* is equal to *coffee* the test case is successful: the verdict **pass** is assigned. Otherwise a failure in the IUT was found: the verdict is **fail**.

As a next step, suppose that the machine cannot be accessed directly by a tester, but that the IUT is embedded in an environment that consists of a butler who operates the machine. The tester has to give orders to the butler, and receives the drinks from the

butler. Giving an order consists of giving a shilling, action *give-shilling*, followed by the question *please* by the butler, and the choice between ordering coffee (*order-coffee*), and ordering tea (*order-tea*). After receiving the order, the butler goes to the machine, puts the *shilling* in, presses the *coffee-button*, respectively the *tea-button*, takes *coffee*, respectively *tea*, and supplies the coffee, *coffee-supply*, or tea *tea-supply*. An abstract test case taking into account this environment of the IUT is the following:

behaviour	constraints	verdict
!*give-shilling*		
?*please*		
!*order-coffee*		
?drink-supply	drink = *coffee*	**pass**
?drink-supply	drink ≠ *coffee*	**fail**

Of course, this only works as a test case for the vending machine if we can assume that the butler, i.e. the test interface, is doing his job correctly.

2.4.2 Formal Specification of the Vending Machine

Now we come to the formal specification of the vending machine, and the formal derivation of test cases according to the methodology discussed in section 2.2. It will be shown that the labelled transition system in figure 2.7 does not uniquely define conforming implementations. Different formal languages $\mathcal{L}_R$ for conformance requirements define different classes of conforming implementations. Indications about what such formal languages should look like are obtained by formalizing the natural language conformance requirements. Ambiguities in these natural language requirements lead to different formalizations, and thus to different languages $\mathcal{L}_R$.

Formalization of Requirements

The first natural language requirement r_1 can be formalized, using the notation for labelled transition systems (table 1.1), as

$$M \stackrel{\mathit{shilling}}{\Longrightarrow} \tag{2.15}$$

However, it is not clear from the natural language specification whether M may accept anything else, e.g. a *penny*. If this is not allowed this can be formalized by

$$M \stackrel{\mathit{penny}}{\Longrightarrow\!\!\!\!\!\!/} \tag{2.16}$$

or, more generally, by the set of requirements

$$\{\ M \stackrel{a}{\Longrightarrow\!\!\!\!\!\!/}\ \mid a \in L,\ a \neq \mathit{shilling}\ \} \tag{2.17}$$

In the second natural language requirement r_2 it is not clear whether the machine allows the environment to choose between pushing the *coffee-button* and the *tea-button*, or whether the machine chooses itself. In the first case the requirement can be formalized by the following two requirements, which shall both be satisfied:

$$\text{if} \quad M \stackrel{\mathit{shilling}}{\Longrightarrow} M' \quad \text{then} \quad M' \stackrel{\mathit{coffee\text{-}button}}{\Longrightarrow} \tag{2.18}$$

$$\text{if} \quad M \stackrel{\mathit{shilling}}{\Longrightarrow} M' \quad \text{then} \quad M' \stackrel{\mathit{tea\text{-}button}}{\Longrightarrow} \tag{2.19}$$

The second case, where M can choose arbitrarily between allowing the *coffee-button* or the *tea-button* to be pushed, can be formalized by:

$$\text{if} \quad M \stackrel{\mathit{shilling}}{\Longrightarrow} M' \quad \text{then} \quad M' \stackrel{\mathit{coffee\text{-}button}}{\Longrightarrow} \ \text{ or } \ M' \stackrel{\mathit{tea\text{-}button}}{\Longrightarrow}$$

or equivalently by:

$$\text{if} \quad M \stackrel{\mathit{shilling}}{\Longrightarrow} M' \quad \text{then} \quad \exists a \in \{\ \mathit{coffee\text{-}button}, \mathit{tea\text{-}button}\ \} : \quad M' \stackrel{a}{\Longrightarrow} \tag{2.20}$$

Neither (2.18) nor (2.19) are conformance requirements in this case. Moreover, analogous to (2.16) requirements like the following could be added:

$$M \stackrel{\mathit{shilling} \cdot \mathit{soup\text{-}button}}{\not\Longrightarrow}$$

Requirement Languages

In the above discussion requirements of the following three forms appear:

$$\stackrel{\sigma}{\Longrightarrow}\ , \quad \stackrel{\sigma}{\not\Longrightarrow}\ , \quad \text{if} \ \stackrel{\sigma}{\Longrightarrow} \ \text{ then } \ \exists a \in A : \stackrel{a}{\Longrightarrow}$$

with $\sigma \in L^*, A \subseteq L$. These three kinds of requirements inspire to the definition of three requirement languages: $\mathcal{L}_{tr}$, $\mathcal{L}_{\overline{tr}}$, and $\mathcal{L}_{must}$. The names will be clarified at the end of this section.

Definition 2.5
The requirement languages $\mathcal{L}_{tr}$, $\mathcal{L}_{\overline{tr}}$, and $\mathcal{L}_{must}$ are defined as:

- $\mathcal{L}_{tr} \ =_{def} \ \{\, \textbf{cannot}\ \sigma \mid \sigma \in L^* \}$
- $\mathcal{L}_{\overline{tr}} \ =_{def} \ \{\, \textbf{can}\ \sigma \mid \sigma \in L^* \}$
- $\mathcal{L}_{must} \ =_{def} \ \{\, \textbf{after}\ \sigma\ \textbf{must}\ A \mid \sigma \in L^*, A \subseteq L \}$

For **spec** and **sat** we take the same relation, indicated by $\models$ and defined as:

- $B \models \textbf{cannot}\ \sigma \ =_{def} \ B \stackrel{\sigma}{\not\Longrightarrow}$
- $B \models \textbf{can}\ \sigma \ =_{def} \ B \stackrel{\sigma}{\Longrightarrow}$
- $B \models \textbf{after}\ \sigma\ \textbf{must}\ A \ =_{def} \ \forall B' (\text{ if } B \stackrel{\sigma}{\Longrightarrow} B' \text{ then } \exists a \in A : B' \stackrel{a}{\Longrightarrow})$

□

Formal Specification and Conformance Requirements

Consider the action tree in figure 2.7 as a behaviour specification $B_M \in \mathcal{LTS}$, and let the label alphabet be given by

$$L = \{\ \textit{shilling, penny, coffee-button, tea-button, soup-button, coffee, tea, soup}\ \}$$

According to equation (2.2) the set of conformance requirements is given by

$$S = \{r \in \mathcal{L}_R \mid B_M\ \textbf{spec}\ r\}$$

For $\mathcal{L}_{\overline{tr}}$ this means that the following finite set of conformance requirements is obtained:

$$\begin{aligned} S_{\overline{tr}} &= \{r \in \mathcal{L}_{\overline{tr}} \mid B_M \models r\} \\ &= \{\ \textbf{can}\ \epsilon\ ,\ \textbf{can}\ \textit{shilling}\ ,\ \textbf{can}\ \textit{shilling}\cdot\textit{coffee-button}\ ,\ \textbf{can}\ \textit{shilling}\cdot\textit{tea-button}\ , \\ &\quad\ \textbf{can}\ \textit{shilling}\cdot\textit{coffee-button}\cdot\textit{coffee}\ ,\ \textbf{can}\ \textit{shilling}\cdot\textit{tea-button}\cdot\textit{tea}\ \} \end{aligned}$$

$S_{\overline{tr}}$ specifies that any trace in the specification must be present in the implementation. It specifies the minimal required behaviour of an implementation. An implementation should at least do what is specified, but it is free to do more. In figure 2.8, I_1, I_2, I_3, and I_4 are conforming implementations of $S_{\overline{tr}}$.

Another class of conforming implementations is defined if $\mathcal{L}_{tr}$ is chosen as $\mathcal{L}_R$. An infinite number of conformance requirements is obtained from B_M:

$$\begin{aligned} S_{tr} &= \{r \in \mathcal{L}_{tr} \mid B_M \models r\} \\ &= \{\ \textbf{cannot}\ \textit{penny}\ ,\ \textbf{cannot}\ \textit{coffee}\ ,\ldots, \\ &\quad\ \textbf{cannot}\ \textit{shilling}\cdot\textit{shilling}\ ,\ \textbf{cannot}\ \textit{shilling}\cdot\textit{soup-button}\ , \\ &\quad\ \textbf{cannot}\ \textit{shilling}\cdot\textit{coffee}\ ,\ldots, \\ &\quad\ \textbf{cannot}\ \textit{shilling}\cdot\textit{coffee-button}\cdot\textit{tea}\ ,\ldots\ \} \\ &= \{\ \textbf{cannot}\ \sigma\ \mid \sigma \notin \textit{traces}(B_M)\ \} \end{aligned}$$

S_{tr} specifies that any trace which is not in the specification, shall not be in the implementation. It specifies the maximal behaviour of an implementation: a conforming implementation may not do more than is allowed in the specification, but doing less is allowed, even doing nothing. In figure 2.8, I_1, I_4, I_5, I_6, I_7, I_8, and I_9 are conforming implementations.

The language $\mathcal{L}_{must}$ takes into account the branching behaviour of a system. It distinguishes between I_1 and I_4 (figure 2.8).

$$\begin{aligned} S_{must} = \{\ & \textbf{after}\ \epsilon\ \textbf{must}\ \{\textit{shilling}\}\ ,\quad \textbf{after}\ \epsilon\ \textbf{must}\ \{\textit{shilling}, \textit{coffee}\}\ ,\ \ldots, \\ & \textbf{after}\ \textit{shilling}\ \textbf{must}\ \{\textit{coffee-button}\}\ , \\ & \textbf{after}\ \textit{shilling}\ \textbf{must}\ \{\textit{tea-button}\}\ , \\ & \textbf{after}\ \textit{shilling}\ \textbf{must}\ \{\textit{coffee-button, tea-button}\}\ ,\ \ldots, \\ & \textbf{after}\ \textit{penny}\ \textbf{must}\ \emptyset\ ,\ \ldots, \\ & \textbf{after}\ \textit{shilling}\cdot\textit{soup}\ \textbf{must}\ \emptyset\ ,\ \ldots\ \} \end{aligned}$$

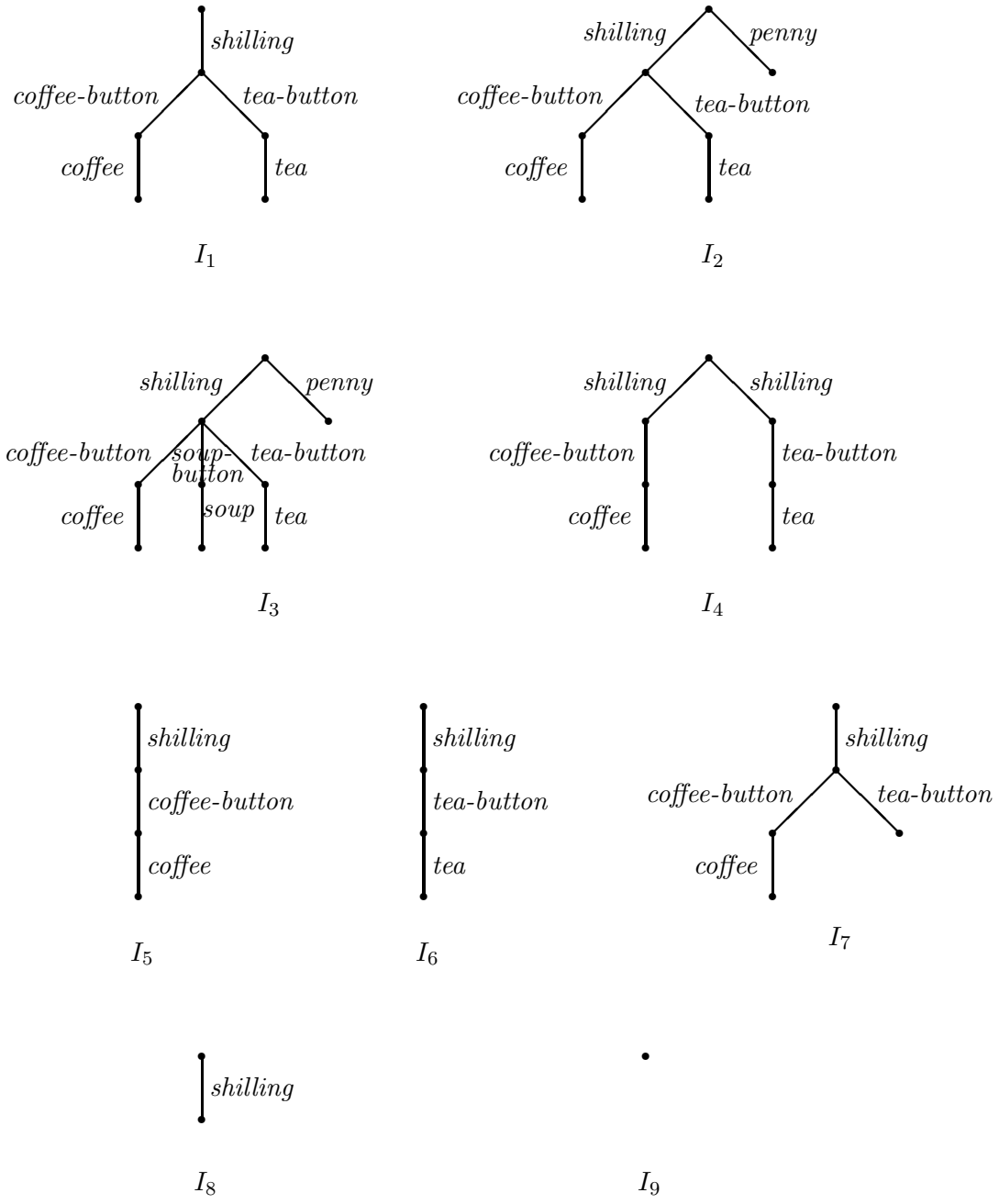


Figure 2.8: Potential implementations.

I_1 is conforming according to $\mathcal{L}_{must}$. It satisfies the requirements

$$\begin{array}{rl} & \textbf{after } \textit{shilling} \textbf{ must } \{\textit{coffee-button}\} \\ \text{and} & \textbf{after } \textit{shilling} \textbf{ must } \{\textit{tea-button}\} \end{array}$$

which hold for the specification. I_4 does not satisfy these requirements, only the requirement

$$\textbf{after } \textit{shilling} \textbf{ must } \{\textit{coffee-button,tea-button}\}$$

It is clear that different choices for the language of requirements $\mathcal{L}_R$ lead to different classes of conforming implementations. A behaviour specification in itself is not sufficient to define what conforming implementations are. A behaviour specification together with a choice for $\mathcal{L}_R$, **spec** and **sat** defines a class of conforming implementations. This choice for $\mathcal{L}_R$ corresponds to a choice for an implementation relation, as explained in section 2.3. The requirement languages $\mathcal{L}_{tr}$ and $\mathcal{L}_{\overline{tr}}$ in this example correspond to the preorders of definition 1.15. According to equation (2.14) $\mathcal{L}_{tr}$ defines the implementation relation $\leq_{tr}$, and $\mathcal{L}_{\overline{tr}}$ defines $\geq_{tr}$. The implementation relation corresponding to the language $\mathcal{L}_{must}$ will be elaborated in chapter 3. It will be shown to correspond to the relation *testing preorder* $\leq_{te}$, or **red** in [Bri88].

Proposition 2.6
Let **conforms-to** $\subseteq \mathcal{LTS} \times \mathcal{LTS}$ be defined by (2.14):

$$B_I \textbf{ conforms-to } B_S \quad =_{def} \quad \forall r \in \mathcal{L}_R : \; B_S \textbf{ spec } r \;\text{ implies }\; B_I \textbf{ sat } r$$

and let $\mathcal{L}_{tr}$, $\mathcal{L}_{\overline{tr}}$, **spec**, and **sat** be given in definition 2.5, then

- For $\mathcal{L}_R = \mathcal{L}_{tr}$: **conforms-to** $= \leq_{tr}$
- For $\mathcal{L}_R = \mathcal{L}_{\overline{tr}}$: **conforms-to** $= \geq_{tr}$

□

By varying the relations **spec** and **sat** even more implementation relations can be obtained. Take as an example the language $\mathcal{L}_{must}$ with **sat** as in definition 2.5 and with **spec** defined by

$$B \;\textbf{ spec }\; \textbf{after } \sigma \textbf{ must } A \quad =_{def} \quad B \;\textbf{ sat }\; \textbf{after } \sigma \textbf{ must } A \;\text{ and }\; \sigma \in \textit{traces}(B)$$

This is an example where **spec** $\subseteq$ **sat**, but not **spec** $\neq$ **sat** (section 2.3, proposition 2.3). The implementation relation defined in this way is indeed reflexive, B **conforms-to** B for any B, but it is not transitive: if B_1 **conforms-to** B_2 and B_2 **conforms-to** B_3 then not necessarily B_1 **conforms-to** B_3. The reader is invited to check in figures 2.8 and 1.7 (section 1.4), that

$$\begin{array}{rlll} & I_7 & \textbf{conforms-to} & I_5 \\ \text{and} & I_5 & \textbf{conforms-to} & \text{figure 1.7} \\ \text{but not} & I_7 & \textbf{conforms-to} & \text{figure 1.7} \end{array}$$

In chapter 3 this implementation relation will be elaborated, and it will be shown to correspond to the relation **conf** of [Bri88] (proposition 3.17).

Many other choices for languages $\mathcal{L}_R$ are possible. Examples are combinations of $\mathcal{L}_{tr}$, $\mathcal{L}_{\overline{tr}}$ and $\mathcal{L}_{must}$, such as $\mathcal{L}_{tr} \cup \mathcal{L}_{\overline{tr}}$. Since (not **can** σ) = **cannot** σ, negation is expressible in $\mathcal{L}_{tr} \cup \mathcal{L}_{\overline{tr}}$, and therefore according to proposition 2.3.4 the corresponding implementation relation is an equivalence. It is *trace equivalence* $\approx_{tr}$ (definition 1.13.4).

An area of interest is the relation between different requirement languages, and hence between different implementation relations. As an example consider the relation between $\mathcal{L}_{tr}$ and $\mathcal{L}_{must}$. It follows from definition 2.5 that for any B

$$B \models \textbf{cannot } \sigma \quad \text{if and only if} \quad B \models \textbf{after } \sigma \textbf{ must } \emptyset$$

Thus any requirement in $\mathcal{L}_{tr}$ can be expressed in $\mathcal{L}_{must}$ (but not vice versa). It follows that any implementation which is correct according to $\mathcal{L}_{must}$ is also correct according to $\mathcal{L}_{tr}$, thus

$$\text{if} \quad I \leq_{te} B \quad \text{then} \quad I \leq_{tr} B$$

Which requirement language is useful in a particular case depends on the application. An example is the difference between the specification of a protocol and a local interface. In a protocol specification it is desirable to specify some *safety* requirements, e.g. a protocol implementation should not send strange messages. In a local interface requirements might only specify minimum behaviour, with freedom to do more, as long as it is local.

From the literature many implementation relations are known. A whole range of them can be defined, each with its own application area, and with relations between them, see e.g. [Lan90, Gla90].

A well-known logical language for labelled transition systems is *Hennessy-Milner Logic* HML [HM85]. The corresponding equivalence is bisimulation equivalence [Par81] defined in 1.13. Sublanguages of HML can be used to specify weaker equivalences e.g. the equivalences used in this section [Lar90]. Related logics with corresponding equivalences are given in [Abr87, Phi87, Sti91].

2.4.3 PICS as a Specification Parameter

Suppose the following modification of the specification of the vending machine M: M *must* supply *coffee*, but supplying *tea* is optional. According to section 1.3.2 it is stated in the *PICS* whether a particular implementation does implement the *tea*-option. The *PICS proforma* could look like

Has the *tea*-option been implemented? : *yes*/*no*.

Formally, this can be conveniently specified by using a *PICS* parameter (section 2.2.2). Figure 2.9 shows a way to represent such a parameterized specification

$$B_M(\textit{PICS-proforma} : \{\textit{yes},\textit{no}\})$$

The transition *tea-button* depends on the condition [*PICS-proforma = yes*]. Only if the condition is fulfilled this transition can take place. The two possible instantiations of this parameterized specification are also given in figure 2.9. Depending on the value of the PICS of a particular IUT either the first, or the second instantiated specification is used as a basis for obtaining the conformance requirements.

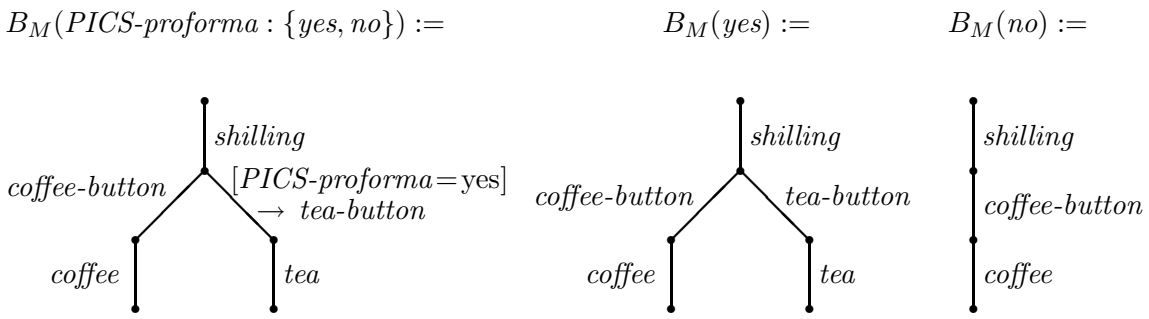


Figure 2.9: Parameterized specification with instantiations.

Another way of specifying optional behaviour is by using a nondeterministic specification. This is shown in figure 2.10. Upon receiving a *shilling* M may choose nondeterministically between making a transition to state s_1 or to state s_2. After *shilling*, M always offers the action *coffee-button*, but the action *tea-button* is only offered in s_1. This means that *tea-button* may be refused after *shilling*.

Specified in $\mathcal{L}_{must}$ we have the requirements:

after ϵ **must** {*shilling*} ,
after *shilling* **must** {*coffee-button*} ,
after *shilling* **must** {*coffee-button, tea-button*} , ...

but not:

after *shilling* **must** {*tea-button*}

A conforming implementation according to $\mathcal{L}_{must}$ must also always offer *coffee-button* after *shilling*, and is allowed to refuse *tea-button*. In figure 2.8, both I_1 and I_5, i.e. $B_M(yes)$ and $B_M(no)$, always offer *coffee-button* after *shilling*; they are conforming implementations. I_4 and I_6 are not conforming: they can refuse *coffee-button* after *shilling*.

Note the difference between these two ways of specifying optional behaviour in case of recursive behaviour. We have recursive behaviour if the machine returns to its initial state after having supplied *coffee* or *tea*, so that it can produce a next drink. In a useful machine, of course, this is a desirable feature. In case of such recursive behaviour the nondeterministic specification allows the implementation to renew its choice each time a *shilling* is supplied. Using the parameterized specification the implementation must choose once and for all, before it starts, whether it will supply *tea* or not.

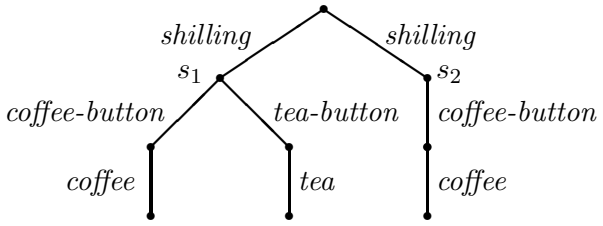


Figure 2.10: Specifying options by nondeterminism.

2.4.4 Application of Logic

In section 2.2.1 it was pointed out that a requirement language can be considered as a logical language, and that specifications, sets of requirements, can be considered as theories. Concepts from logic can be defined for requirement languages. As an example a few logical concepts are defined for the language $\mathcal{L}_{\overline{tr}}$. The presentation is intended for illustration, not for practical use. We start with defining *derivation* for $\mathcal{L}_{\overline{tr}}$.

Derivation for a logical language is introduced by defining *axioms* and *inference rules*. An axiom is a formula that is assumed to hold. An inference rule defines how a formula can be obtained from other formulae. Let $q_1, q_2, \ldots, q_n \vdash q$ be an inference rule, then, if we can prove $q_1, q_2, \ldots, q_n$ (the premisses), then we may conclude q (the conclusion).

A *derivation* for a formula r from a given set of formulae S is a finite sequence $r_1, r_2, \ldots, r_n$ of formulae such that

- the last element r_n of the sequence is equal to r; and
- for each r_i in the sequence: either r_i is an axiom, or $r_i \in S$, or there is an inference rule $r_{j_1}, r_{j_2}, \ldots, r_{j_m} \vdash r_i$, such that $r_{j_1}, r_{j_2}, \ldots, r_{j_m}$ are elements of the sequence preceding r_i $(j_1, j_2, \ldots j_m < i)$.

If there is derivation for r from S this is written $S \vdash r$.

For the requirement language $\mathcal{L}_{\overline{tr}}$ an axiom is

$$\vdash \quad \mathbf{can}\ \epsilon \tag{2.21}$$

It expresses the intuition that any system can always do nothing.

An inference rule is, for $\sigma_1, \sigma_2 \in L^*$:

$$\mathbf{can}\ \sigma_1 \cdot \sigma_2 \quad \vdash \quad \mathbf{can}\ \sigma_1 \tag{2.22}$$

which expresses the intuition that any system that can perform a trace, can also perform a prefix of that trace.

Using this axiom and this inference rule derivation is simplified due to the fact that a prefix of a prefix of a trace is also a prefix of the original trace, so that any derivation

can always be made in one step. Hence, a requirement $r = \mathbf{can}\ \sigma\ \in \mathcal{L}_{\overline{tr}}$ can be derived from a specification $S \subseteq \mathcal{L}_{\overline{tr}}$, $S \vdash \mathbf{can}\ \sigma$, if and only if

$$\mathbf{can}\ \sigma\ \in S;\ \text{ or} \tag{2.23}$$

$$\mathbf{can}\ \sigma\ \text{ is an axiom} : \sigma = \epsilon;\ \text{ or} \tag{2.24}$$

$$\mathbf{can}\ \sigma\ \text{ is a consequence of (2.22)} : \exists \sigma'\ \text{ such that }\ \mathbf{can}\ \sigma \cdot \sigma' \in S. \tag{2.25}$$

Derivation is only interesting if it is *sound* with respect to a satisfaction relation $\models$, which means that if r can be derived from S then every model of S is also model of r:

$$\text{if}\ \ S \vdash r\ \ \text{then}\ \ \forall I\ (\ I \models S\ \ \text{implies}\ \ I \models r\ ) \tag{2.26}$$

Derivation is *complete* if the converse holds.

Soundness means that only true formulae can be derived. Completeness means that there are sufficient axioms and inference rules to derive all true formulae. For (2.23), (2.24), and (2.25) we have the following:

Proposition 2.7
Derivation for $\mathcal{L}_{\overline{tr}}$ as defined by (2.23), (2.24), and (2.25) is sound and complete. □

Derivation is useful in testing, since it can reduce the number of relevant conformance requirements. Requirements that can be derived from other requirements are superfluous. A theory that cannot be further reduced without affecting its meaning, is called *logically independent*:

$$S \subseteq \mathcal{L}_R\ \text{ is } \textit{logically independent} \quad =_{def} \quad \text{not}\ \ \exists r \in S : S \backslash \{r\} \vdash r \tag{2.27}$$

In a sound and complete derivation system this means that none of the requirements can be removed without changing the class of conforming implementations:

$$S\ \text{ is logically independent}\ \ \text{iff}\ \ \forall r \in S, \exists I \in \mathcal{L}_{FDI} : I \not\models S\ \ \text{and}\ \ I \models S \backslash \{r\} \tag{2.28}$$

For $S_{\overline{tr}}$ we have that $\mathbf{can}\ \epsilon$ is an axiom, and thus it holds for any implementation. Testing for such a requirement does not make sense. Moreover,

$$\mathbf{can}\ \textit{shilling} \cdot \textit{coffee-button} \cdot \textit{coffee} \ \vdash\ \mathbf{can}\ \textit{shilling} \cdot \textit{coffee-button} \ \vdash\ \mathbf{can}\ \textit{shilling}$$

and analogously for the *tea*-branch. This implies that $S_{\overline{tr}}$ can be reduced to $S'_{\overline{tr}}$ with

$$S'_{\overline{tr}} \ = \ \{\ \mathbf{can}\ \textit{shilling} \cdot \textit{coffee-button} \cdot \textit{coffee}\ ,\ \mathbf{can}\ \textit{shilling} \cdot \textit{tea-button} \cdot \textit{tea}\ \}$$

$S'_{\overline{tr}}$ is logically independent. It cannot be further reduced. Note that for $\mathcal{L}_{\overline{tr}}$ not always such a reduced, independent specification can be found. An example is a system that can do a infinitely many often: $B = a; B$. Non-existence of an independent specification is a result of the fact that $\langle L^*, \preceq \rangle$ is not co-well-founded (cf. appendix A): because we

can make infinite sequences of preceding traces 'to the right', we can make infinite derivations 'to the left', using (2.22).

If a theory S is extended with all requirements that can be derived from it using $\vdash$, we get the *deductive closure*, denoted by $\overline{S}$. For our example $S_{\overline{tr}}$ is the deductive closure of $S'_{\overline{tr}}$.

$$\overline{S} = \{r \in \mathcal{L}_R \mid S \vdash r\} \tag{2.29}$$

Due to the soundness (2.26) of the axiom and inference rule $S'_{\overline{tr}}$ allows exactly the same implementations as $S_{\overline{tr}} = \overline{S'_{\overline{tr}}}$.

Another concept is *consistency* of a specification. A specification S is consistent if no contradiction can be derived from it. If derivation is sound and complete this means that there is at least one implementation that satisfies all requirements in S. By construction of $\mathcal{L}_{\overline{tr}}$ any specification in it is consistent: the implementation that can always perform every action in L satisfies any $S \subseteq \mathcal{L}_{\overline{tr}}$. On the other hand a specification in $\mathcal{L}_{tr}$ can be inconsistent, e.g. there is no implementation that satisfies $S = \{$ **cannot** $\epsilon \}$.

A specification S is syntactically *complete* if each requirement or its negation can be derived from it. For sound and complete derivation this implies that there exists at most one implementation that satisfies S. Logical completeness of specifications is mostly not required; specifications leave some freedom for the implementer. This was already illustrated by the specifications S_{tr} and $S_{\overline{tr}}$ in the example.

2.4.5 Test Generation

Test Purposes

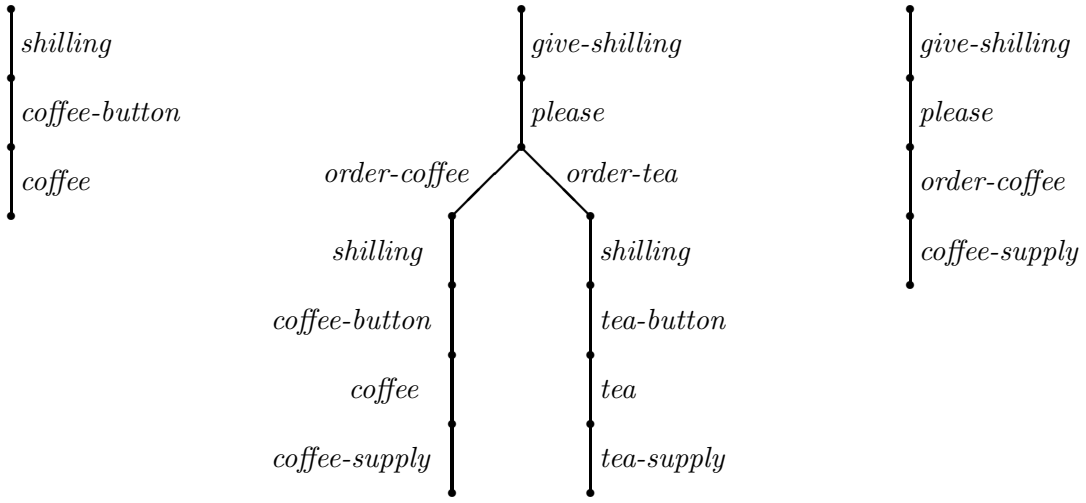
The first step in deriving abstract test cases is the identification of test purposes. Formally, this means a selection of conformance requirements (section 2.2.3). $S_{\overline{tr}}$ contains 6 requirements, which can be reduced to the two requirements of $S'_{\overline{tr}}$. This means that exhaustive testing in the generic sense is possible.

Generic Test Cases

Take the test notation $\mathcal{L}_T$ to be $\mathcal{LTS}$, then a generic test case corresponding to the test purpose **can** *shilling·coffee-button·coffee* is the labelled transition system that can only execute the trace *shilling· coffee-button· coffee* (figure 2.11(a)).

More generally, if **can** $\sigma \in \mathcal{L}_{\overline{tr}}$ is a test purpose, then g_σ, the action tree consisting of the sequence of actions σ, is valid with respect to **can** σ . Application of test case g_σ to implementation I consists of repeatedly running g_σ and I in parallel, interacting synchronously with each other.

This can be elaborated formally in the notation of section 2.2.4 as follows.



(a) Generic test case. *(b) The butler.* *(c) Abstract test case.*

Figure 2.11:

Define $g_\sigma \in \mathcal{LTS}$ by (definitions 1.6 and 1.7):

$$\begin{cases} g_\epsilon & =_{def} & \mathbf{stop} \\ g_{a\cdot\sigma} & =_{def} & a; g_\sigma \end{cases}$$

The observations that we can make by testing an implementation I with a generic test case g are the traces that both can perform running synchronously in parallel. Thus $\Omega = L^*$ and $run \subseteq \mathcal{LTS} \times \mathcal{LTS} \times L^*$ is

$$run(g, I, \sigma) \quad =_{def} \quad g \,\|\, I \stackrel{\sigma}{\Longrightarrow}$$

and thus $runs : \mathcal{LTS} \times \mathcal{LTS} \to \mathcal{P}(L)$ is

$$runs(g, I) \;=\; \{\; \sigma \in L^* \mid g \,\|\, I \stackrel{\sigma}{\Longrightarrow} \} \tag{2.30}$$

The application of a test g_σ to an implementation I is successful if the trace σ can be observed at least once. The function $eval : \mathcal{P}(L) \to \{\mathbf{pass}, \mathbf{fail}\}$ is defined by

$$eval(runs(g_\sigma, I)) \quad =_{def} \quad \begin{cases} \mathbf{pass} & \text{if } \sigma \in runs(g_\sigma, I) \\ \mathbf{fail} & \text{if } \sigma \notin runs(g_\sigma, I) \end{cases} \tag{2.31}$$

Now we can prove the validity according to equation (2.10). For each $I \in \mathcal{LTS}$ we have:

$apply(g_\sigma, I) = \mathbf{pass}$	iff	$(*$ equation (2.12) $*)$
$eval(runs(g_\sigma, I)) = \mathbf{pass}$	iff	$(*$ equation (2.31) $*)$
$\sigma \in runs(g_\sigma, I)$	iff	$(*$ equation (2.30) $*)$
$g_\sigma \parallel I \stackrel{\sigma}{\Longrightarrow}$	iff	$(*$ definition 1.7 $*)$
$g_\sigma \stackrel{\sigma}{\Longrightarrow}$ and $I \stackrel{\sigma}{\Longrightarrow}$	iff	$(*$ by construction of g_σ, $g_\sigma \stackrel{\sigma}{\Longrightarrow}$ always holds $*)$
$I \stackrel{\sigma}{\Longrightarrow}$	iff	$(*$ definition 2.5 $*)$
I **sat** **can** σ		

This concludes the proof that for each σ, g_σ is a valid generic test case with respect to the test purpose **can** σ .

Abstract Test Cases

We return to the situation that the vending machine cannot be accessed directly by a tester, but that it is embedded in an environment that consists of a butler who operates the machine. The butler forms the *test interface* of section 2.2.6. In order to generate abstract test cases formally, we must have a formal description of the butler. Figure 2.11(b) gives such a description. The implementation access points (*IAPs*) are *shilling, coffee-button, coffee, tea-button, tea.* Via *IAPs* communication between the test interface (the butler) and the IUT takes place. The points of control and observation (*PCOs*) are *give-shilling, please, order-coffee, order-tea, coffee-supply, tea-supply.* Via *PCOs* the tester communicates with the test interface, and in this way indirectly with the IUT. There is no physical distribution of testing functions in this example, and all *IAPs* are indirectly accessible. It can easily be checked that the abstract test case of figure 2.11(c) corresponds to the generic test case of figure 2.11(a).

2.5 Summary of the Testing Framework

The use of formal methods in conformance testing has been approached by giving a formal interpretation of concepts in the informal standard ISO9646. It was not the intention to present new theories, but to relate existing approaches in the practice of conformance testing to existing theories in the realm of processes and concurrency, resulting in a framework for formal conformance testing. Since not all presented formalizations will be used in the next chapters, this section summarizes the most important concepts that will be used. Moreover, some additional notation is introduced.

We distinguished between two approaches of specifying distributed systems, viz. the logical approach and the behavioural approach. A logical specification consists of a set of formulae, or requirements, and conformance is expressed by satisfaction of these requirements. A behaviour specification defines the observable behaviour of a system, and conformance is expressed by a relation.

In the next chapters the starting point for studying conformance will be a behavioural specification technique $\mathcal{L}_{FDT}$. More specifically it will be $\mathcal{LTS}$ in most cases. We assume that implementations can be modelled by the same specification formalism: $\mathcal{L}_{IMPL} = \mathcal{L}_{FDT}$, so that conformance is expressed by a (preorder) relation on $\mathcal{L}_{FDT}$. We refer to this relation as the *implementation relation* (section 2.3), and denote it by $\leq_{\mathcal{R}}$.

Testing assumes the existence of a universe of test cases. This is modelled by a *test notation* $\mathcal{L}_T$. Application of a test case $t \in \mathcal{L}_T$ to an implementation $I \in \mathcal{L}_{FDT}$ is modelled by the function $apply : \mathcal{L}_T \times \mathcal{L}_{FDT} \rightarrow \{\mathbf{pass}, \mathbf{fail}, \mathbf{inconclusive}\}$.

It is possible to identify the verdicts **pass** and **inconclusive**. This allows to abstract from the function *apply*, and to replace it by a relation $\mathbf{passes} \subseteq \mathcal{L}_{FDT} \times \mathcal{L}_T$:

$$I \ \mathbf{passes}\ t \quad =_{def} \quad apply(t, I) \neq \mathbf{fail}$$

This simplification is justified by section 2.2.8: we can only demonstrate the presence of errors, not their absence. The distinction between **pass** and **inconclusive** makes only sense when a test case is related to an individual test purpose, not for correctness of implementations with respect to specifications.

The relation **passes** is straightforwardly extended to test suites, i.e. sets of test cases $\Pi \subseteq \mathcal{L}_T$. We introduce the notation **fails** as its negation.

Notation 2.8
Let $I \in \mathcal{L}_{FDT}$, $\Pi \in \mathcal{P}(\mathcal{L}_T)$, $t \in \mathcal{L}_T$:

1. $I \ \mathbf{passes}\ \Pi \quad =_{def} \quad \forall t \in \Pi : I \ \mathbf{passes}\ t$
2. $I \ \mathbf{fails}\ t \quad =_{def} \quad I \ \mathbf{passes}\!\!\!/\ t$
3. $I \ \mathbf{fails}\ \Pi \quad =_{def} \quad I \ \mathbf{passes}\!\!\!/\ \Pi$

□

Test derivation consists of systematically deriving a test suite from $S \in \mathcal{L}_{FDT}$, with the intent of testing implementations for correctness with respect to an implementation relation $\leq_{\mathcal{R}}$. It can be expressed as a function $\Pi_{\mathcal{R}} : \mathcal{L}_{FDT} \rightarrow \mathcal{P}(\mathcal{L}_T)$.

If a test suite is derived according to the methodology of section 2.2, it has the property that only incorrect implementation are rejected. Such test suites are called *sound*. If a test suite rejects all incorrect implementations, and possibly more, it is called *exhaustive*. A test suite that rejects all and only incorrect implementations is called *complete*.

Definition 2.9
Let $\mathcal{L}_{FDT}$ be a behavioural specification formalism, $\leq_{\mathcal{R}} \subseteq \mathcal{L}_{FDT} \times \mathcal{L}_{FDT}$ an implementation relation, $\mathcal{L}_T$ a test notation, and $\mathbf{passes} \subseteq \mathcal{L}_{FDT} \times \mathcal{L}_T$, then

1. A *test derivation* is a function $\Pi_{\mathcal{R}} : \mathcal{L}_{FDT} \rightarrow \mathcal{P}(\mathcal{L}_T)$.
2. Let $\Pi \in \mathcal{P}(\mathcal{L}_T)$ be a test suite for S with respect to $\leq_{\mathcal{R}}$, then
   - Π is *sound* $\quad =_{def} \quad \forall I \in \mathcal{L}_{FDT} : I \leq_{\mathcal{R}} S$ implies $I \ \mathbf{passes}\ \Pi$
   - Π is *exhaustive* $\quad =_{def} \quad \forall I \in \mathcal{L}_{FDT} : I \not\leq_{\mathcal{R}} S$ implies $I \ \mathbf{fails}\ \Pi$

- Π is *complete* $=_{def}$ Π is both exhaustive and sound.

3. A test derivation $\Pi_{\mathcal{R}}$ is sound, exhaustive, or complete, if for all S, $\Pi_{\mathcal{R}}(S)$ is sound, exhaustive, or complete respectively.

□

If the logical approach is used, a specification consists of a set of requirements, which corresponds to a logical theory. Given a logical specification $S \subseteq \mathcal{L}_R$, and the class of models $\mathcal{L}_{FDT}$, conformance is expressed using a satisfaction relation $\mathbf{sat} \subseteq \mathcal{L}_{FDT} \times \mathcal{L}_R$ (equation (2.1)):

$$I \ \mathbf{sat}\ S \quad =_{def} \quad \forall r \in S : \ I \ \mathbf{sat}\ r$$

Apart from the specification, also an implementation can be related to a theory: the theory of all requirements that it satisfies, denoted by $sats(I)$:

$$sats(I) \quad =_{def} \quad \{\ r \in \mathcal{L}_R \mid I \ \mathbf{sat}\ r\ \}$$

In this setting conformance is expressed by requiring that the theory S is a subtheory of $sats(I)$:

$$S \ \subseteq \ sats(I)$$

The relation between behaviour specifications and logical specifications is expressed by a relation $\mathbf{spec} \subseteq \mathcal{L}_{FDT} \times \mathcal{L}_R$. Using **spec** the theory specified by $B_S \in \mathcal{L}_{FDT}$, denoted by $specs(B_S)$, is defined by (equation (2.2)):

$$specs(B_S) \quad =_{def} \quad \{\ r \in \mathcal{L}_R \mid B_S \ \mathbf{spec}\ r\ \}$$

Now we can state that a behavioural approach of defining conformance is compatible with a logical approach, if implementations conforming according to $\leq_{\mathcal{R}}$, also logically conform to the theory $specs(B_S)$. This is just another way of expressing equation (2.14).

Definition 2.10
An implementation relation $\leq_{\mathcal{R}} \subseteq \mathcal{L}_{FDT} \times \mathcal{L}_{FDT}$ is *compatible* with a requirement language $\mathcal{L}_R$, with relations $\mathbf{sat}, \mathbf{spec} \subseteq \mathcal{L}_{FDT} \times \mathcal{L}_R$, if for all $B_S, I \in \mathcal{L}_{FDT}$:

$$I \leq_{\mathcal{R}} B_S \quad \text{iff} \quad I \ \mathbf{sat}\ specs(B_S)$$

□

Finally, we can identify the theory of a test suite Π as those requirements that are tested by Π, i.e. the requirements that are satisfied by all and only implementations that pass Π. These requirements are denoted by $testreqs(\Pi)$:

$$\forall I \in \mathcal{L}_{FDT} : \quad I \ \mathbf{passes}\ \Pi \quad \text{iff} \quad I \ \mathbf{sat}\ testreqs(\Pi) \tag{2.32}$$

In this setting a sufficient condition for completeness of a test suite Π is that the requirements tested by Π are exactly those contained in S:

$$S \ = \ testreqs(\Pi) \tag{2.33}$$

Now that we have a framework for conformance testing the next step is to fill it in with specific implementation relations, test derivation techniques, etc. This is done in the following chapters.

Chapter 3

Implementation Relations

3.1 Introduction

Conformance can be expressed by means of an implementation relation (sections 2.3 and 2.5). Given a specification formalism $\mathcal{L}_{FDT}$ and assuming that implementations can be modelled by elements of $\mathcal{L}_{FDT}$ an implementation relation $\leq_{\mathcal{R}} \subseteq \mathcal{L}_{FDT} \times \mathcal{L}_{FDT}$ expresses the correctness of implementations with respect to specifications: $I \leq_{\mathcal{R}} S$ if and only if I is a correct implementation of the behaviour specification S. A natural question now is which relations are appropriate to be implementation relations. A related question is what logical languages are suitable to describe requirements, and how these requirements are specified and satisfied.

This chapter approaches this question by reversing the test case generation problem. Instead of deriving a test case $t \in \mathcal{L}_T$ for a conformance requirement, such that from the observations of testing an implementation I with t satisfaction of the requirement can be concluded, we study the properties that can be tested when a class of test cases is given. Such testable properties give a natural way of defining a notion of equivalence for specifications and implementations: two systems are equivalent if they satisfy the same testable properties. This principle of defining an equivalence can be applied to various specification formalisms. By varying the class of test cases different equivalences are obtained.

The main attention of this chapter is on such equivalences and related implementation relations for labelled transition systems. Such relations were introduced in [DNH84, DN87], and called *testing equivalences*. The result of this chapter is an instantiation of the formal testing framework of chapter 2 (section 2.5) with the specification formalism of labelled transition systems $\mathcal{LTS}$ and implementation relations, corresponding requirement languages, test notation languages, and test application functions. A basic assumption in this chapter is that the test case and the implementation communicate directly, without a test interface or test context, hence this chapter is restricted to generic testing (section 2.2.5).

The next section starts with a discussion of testing equivalence in general, followed by testing equivalence for labelled transition systems. A logical language for requirements that characterizes this equivalence is introduced. In section 3.3 implementation relations for labelled transition systems are defined. One of these relations, the implementation relation **conf**, is elaborated in section 3.4.

3.2 Testing Equivalence

In section 2.2.4 the problem of test case generation was described as finding a test case $t_i \in \mathcal{L}_T$ for each test purpose p_i such that from the observations of testing I with t_i satisfaction of p_i can be concluded. In search for a suitable equivalence and implementation relations, or, equivalently, a suitable requirement language $\mathcal{L}_R$, this statement is reversed: given a class of test cases which requirements or properties can be tested?

Let $\mathcal{L}_T$ be the class of test cases that can be executed, then a conformance requirement or test purpose does not make sense if it cannot be tested by a test $t \in \mathcal{L}_T$, i.e. if there is no test in $\mathcal{L}_T$ such that from the observations made during testing we can conclude about satisfaction of that property. For such an untestable property we cannot distinguish between two systems one of which satisfies the property and the other does not, so an implementation cannot be rejected as non-conforming based on such a property. Two systems that do not differ in any testable property cannot be distinguished at all, so they are equivalent.

Let the observations made by testing an implementation $I \in \mathcal{L}_{FDT}$ with a test $t \in \mathcal{L}_T$ be given by $runs(t, I)$ (section 2.2.4), then the systems I_1 and I_2 are *testing equivalent with respect to* $\mathcal{L}_T$, $I_1 \approx_T I_2$, if for all possible tests that can be performed, the observations made of I_1 and I_2 are the same:

$$I_1 \approx_T I_2 \quad =_{def} \quad \forall t \in \mathcal{L}_T : runs(t, I_1) = runs(t, I_2) \tag{3.1}$$

The class $\mathcal{L}_T$ of test cases defines testing equivalence on $\mathcal{L}_{FDT}$ in a natural way. Different classes of tests define different equivalences. The more powerful the tests in $\mathcal{L}_T$ are, the more discriminating properties can be observed, hence more systems can be distinguished, and a finer equivalence is obtained.

A specific choice for the class of test cases $\mathcal{L}_T$ can be made if we consider the kind of systems that we are testing. These systems are usually components of distributed systems, e.g. protocol entities. A specification describes the external behaviour of such a system, i.e. how a system is observed by its environment. The internal functioning does not matter; the system is considered as a black box. A consequence is that the environment determines which properties are important to be observed.

For components of distributed systems the natural environment consists of other components. These components are systems of the same kind, and can be modelled by

the same formalism. For example the environment of an (N)-protocol entity consists of the (N-1)- and the (N+1)-protocol entities, which can all be described by the same behavioural specification formalism. This means that if our systems are modelled by $\mathcal{L}_{FDT}$, also the environment is modelled by $\mathcal{L}_{FDT}$, and hence the behaviour of test cases should be described in $\mathcal{L}_{FDT}$.

Choosing $\mathcal{L}_T = \mathcal{L}_{FDT}$ in (3.1) we obtain the definition of *testing equivalence on* $\mathcal{L}_{FDT}$:

$$I_1 \approx_{FDT} I_2 \quad =_{def} \quad \forall t \in \mathcal{L}_{FDT} : runs(t, I_1) = runs(t, I_2) \tag{3.2}$$

Testing Equivalence for Labelled Transition Systems

Applying the principle of testing equivalence (3.2) to the specification formalism of labelled transition systems (section 1.4) we have for $I_1, I_2 \in \mathcal{LTS}$:

$$I_1 \approx_{\mathcal{LTS}} I_2 \quad =_{def} \quad \forall t \in \mathcal{LTS} : runs(t, I_1) = runs(t, I_2) \tag{3.3}$$

This leaves open what a test run is, and what kind of observations can be extracted by testing a labelled transition system I with a labelled transition system t.

During a test run t and I communicate. We model this communication by the *synchronous interaction* of t and I: I can perform an observable action $a \in L$ if and only if t performs the same action a. In labelled transition systems this is formalized by the synchronization operator $\|$: $\mathcal{LTS} \times \mathcal{LTS} \to \mathcal{LTS}$ (definitions 1.6 and 1.7):

$$t \parallel I$$

Other kinds of interaction are possible, e.g. asynchronous communication via queues or via buffers. This kind of communication can be modelled by introducing a test interface (section 2.2.6). It is elaborated for labelled transition systems in chapter 5, where it is shown that the kind of interaction that is assumed between t and I influences the resulting testing equivalence.

If test execution is modelled by $t \parallel I$ the observations that can be considered are the occurrence and the non-occurrence of (sequences of) observable actions:

$$t \parallel I \stackrel{\sigma}{\Longrightarrow} \qquad \text{and} \qquad \text{if } t \parallel I \stackrel{\sigma}{\Longrightarrow} B \text{ then } B \stackrel{a}{\Longrightarrow\!\!\!\!\!\!/} \text{ for all } a \in L$$

These notions of observation are formalized by two sets of traces, one set with traces after which no action can be observed (*deadlocks*), and one set with traces that can be observed. Thus the definition of testing equivalence for labelled transition systems $\approx_{te}$ is obtained, instantiating (3.3) with $Obs(t, I)$ and $Obs'(t, I)$ for $runs(t, I)$.

Definition 3.1

1. I **after** σ **deadlocks** $\quad =_{def} \quad \exists I' \in I \textbf{ after } \sigma : \forall a \in L : I' \stackrel{a}{\Longrightarrow\!\!\!\!\!\!/}$
2. The observation functions $Obs, Obs' : \mathcal{LTS} \times \mathcal{LTS} \to \mathcal{P}(L^*)$ are defined by

- $Obs(t, I) \;=_{def}\; \{\sigma \in L^* \mid t \,\|\, I \textbf{ after } \sigma \textbf{ deadlocks}\}$
- $Obs'(t, I) \;=_{def}\; \{\sigma \in L^* \mid t \,\|\, I \stackrel{\sigma}{\Longrightarrow}\}$

3. Testing equivalence on labelled transition systems $\approx_{te} \subseteq \mathcal{LTS} \times \mathcal{LTS}$ is defined by

$$I_1 \approx_{te} I_2 \;=_{def}\; \forall t \in \mathcal{LTS} : Obs(t, I_1) = Obs(t, I_2) \text{ and } Obs'(t, I_1) = Obs'(t, I_2)$$

□

This *extensional* definition of $\approx_{te}$, i.e. this definition in terms of observations by an environment (test cases), can be simplified (proposition 3.3), and rewritten to an *intensional* characterization, i.e. a characterization in terms of properties of the labelled transition systems I_1 and I_2 themselves (theorem 3.6).

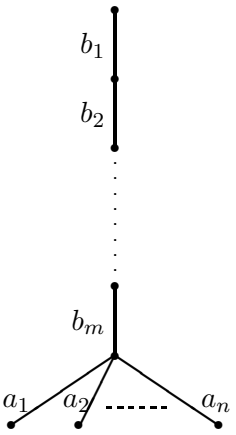
The first simplification is that the observations $Obs'(t, I)$ are superfluous (proposition 3.3.1). The second simplification requires the definition of a subclass of test cases $\mathcal{LT}_M$, the *must tests*. This subclass appears to be sufficient to characterize $\approx_{te}$.

Definition 3.2
Let $\sigma \in L^*$, $A \subseteq L$, then the *must test* $t_{[\sigma,A]}$ is defined inductively by

$$\begin{cases} t_{[\epsilon,A]} & =_{def} \quad \sum\{a; \textbf{stop} \mid a \in A\} \\ t_{[b\cdot\sigma,A]} & =_{def} \quad b; t_{[\sigma,A]} \end{cases}$$

$\mathcal{LT}_M \subseteq \mathcal{LTS}$ is the class of must tests: $\mathcal{LT}_M \;=_{def}\; \{t_{[\sigma,A]} \mid \sigma \in L^*, A \subseteq L\}$

□



Must test $t_{[\sigma,A]}$
with $\sigma = b_1 \cdot b_2 \cdot \ldots \cdot b_m$
and $A = \{a_1, a_2, \ldots, a_n\}$

Figure 3.1: A must test.

Proposition 3.3

1. $I_1 \approx_{te} I_2$ iff $\forall t \in \mathcal{LTS} : Obs(t, I_1) = Obs(t, I_2)$
2. $I_1 \approx_{te} I_2$ iff $\forall t \in \mathcal{LT}_M : Obs(t, I_1) = Obs(t, I_2)$

□

A must test $t_{[\sigma,A]}$ deadlocks with an implementation I either during execution of σ, or if the trace σ is executed and I cannot perform any of the actions in A, or if the must test reaches a terminal state after having executed σ and an $a \in A$ (figure 3.1). Consider the second possibility that can cause deadlock: it points out that I *may* refuse to perform any of the actions in A after having performed σ. This inspires to define the requirement language $\mathcal{L}_{ref}$ reflecting this property. A property of the form **after** σ **refuses** A expresses that it is possible (but not necessary) for I to deadlock for all actions in A after it has executed the trace σ. The negation of the property **after** σ **refuses** A turns out to be exactly a property in $\mathcal{L}_{must}$, which was already introduced in the examples of section 2.4. A property in $\mathcal{L}_{must}$ of the form I **after** A **must** σ expresses that I after having executed the sequence of actions σ is always able to execute at least one of the actions in the set A. A must test $t_{[\sigma,A]}$ exactly tests whether **after** σ **must** A is satisfied: proposition 3.5.6.

Definition 3.4
The requirement languages $\mathcal{L}_{ref}$ and $\mathcal{L}_{must}$ are defined as:

- $\mathcal{L}_{ref} =_{def} \{$ **after** σ **refuses** $A \mid \sigma \in L^*, A \subseteq L\}$
- $\mathcal{L}_{must} =_{def} \{$ **after** σ **must** $A \mid \sigma \in L^*, A \subseteq L\}$

Satisfaction of requirements is defined by:

- I **sat** **after** σ **refuses** $A =_{def} \exists I' \in I$ **after** $\sigma : \forall a \in A : I' \stackrel{a}{\Longrightarrow\!\!\!\!\!/}$
- I **sat** **after** σ **must** $A =_{def} \forall I' \in I$ **after** $\sigma : \exists a \in A : I' \stackrel{a}{\Longrightarrow}$

For short we write I **after** σ **refuses** A and I **after** σ **must** A, respectively.

□

Proposition 3.5

1. I **after** σ **refuses** A iff not (I **after** σ **must** A)
2. if I **after** σ **refuses** A_1 and $A_1 \supseteq A_2$ then I **after** σ **refuses** A_2
3. if I **after** σ **must** A_1 and $A_1 \subseteq A_2$ then I **after** σ **must** A_2
4. I **after** σ **deadlocks** iff I **after** σ **refuses** L
5. I **after** σ **refuses** $\emptyset$ iff $\sigma \in traces(I)$
6. I **after** σ **refuses** A iff $\sigma \in Obs(t_{[\sigma,A]}, I)$

□

Now it follows that $\approx_{te}$ is fully characterized by properties of the form **after** σ **must** A or of the form **after** σ **refuses** A. This gives the intensional characterization of $\approx_{te}$.

Theorem 3.6

$$I_1 \approx_{te} I_2 \quad \text{iff} \quad \forall \sigma \in L^*, \forall A \subseteq L : I_1 \textbf{ after } \sigma \textbf{ must } A \quad \text{iff} \quad I_2 \textbf{ after } \sigma \textbf{ must } A$$

□

Testing equivalence $\approx_{te}$ is the finest equivalence on $\mathcal{LTS}$ for which differences between non-equivalent processes can be observed with our notion of observation, i.e. testers in $\mathcal{LTS}$, using synchronous communication, and observing traces (Obs') and deadlocks (Obs). That traces can be observed is clear: they consist of actions that are observable by definition. Observation of deadlock requires some argumentation. Deadlock, i.e. the non-occurrence of actions, is not observable, only the occurrence of actions can be observed. If no action occurs it might be caused by a long delay: transition systems do not impose a maximum delay on the occurrence of actions, so there can never be certainty that actions are really refused. This dilemma can be resolved by assuming that in practice very long delays are unacceptable, and can be identified with deadlock.

Testing equivalence $\approx_{te}$ will be our basic notion of equality of processes, which in fact means that we do not model systems by labelled transitions systems, but by equivalence classes of $\mathcal{LTS}/\approx_{te}$.

We end this section with a comparison of testing equivalence with the equivalences defined in section 1.4, extending proposition 1.14.2.

Proposition 3.7
$\equiv \subset \sim \subset \approx \subset \approx_{te} \subset \approx_{tr}$

□

3.3 Implementation Relations

An implementation relation formalizes the notion of correctness of an implementation I with respect to a specification S. Analogous to testing equivalence, implementation relations can be obtained by comparing observations made of I with observations made of S. Unlike testing equivalence equality of observations is not required. For an implementation relation it is sufficient that observations of the implementation can be related to observations of the specification, in the sense that the behaviour of the implementation can be 'explained' from the behaviour of the specification. An implementation is considered correct if all observations made of the implementation by any environment can be explained from the behaviour of the specification.

To define implementation relations for labelled transition systems this principle is applied to the observations Obs and Obs' introduced in the previous section: traces observed by testing an implementation I with a test case t should also be observed by testing S with t; and deadlocks observed by testing I with t should also be observed by testing S with t. The resulting implementation relation turns out to be the well-known relation *testing preorder*, or *failure preorder* $\leq_{te}$ (*reduction* **red** in [BSS87]).

Definition 3.8
Let $I, S \in \mathcal{LTS}$, then

$$I \leq_{te} S \quad =_{def} \quad \forall t \in \mathcal{LTS} : Obs(t, I) \subseteq Obs(t, S) \text{ and } Obs'(t, I) \subseteq Obs'(t, S)$$

□

Completely analogous to testing equivalence, this extensional definition can be rewritten to an intensional characterization: the observations in $Obs'(t, I)$ are superfluous, must tests can be used to give a complete characterization, and finally $\leq_{te}$ can be characterized using properties of the forms **after** σ **refuses** A or **after** σ **must** A.

Theorem 3.9

$$\begin{array}{lll} I \leq_{te} S & \text{iff} & \forall t \in \mathcal{LTS} : Obs(t, I) \subseteq Obs(t, S) \\ & \text{iff} & \forall t \in \mathcal{LT}_M : Obs(t, I) \subseteq Obs(t, S) \\ & \text{iff} & \forall \sigma \in L^*, \forall A \subseteq L : \\ & & \quad I \textbf{ after } \sigma \textbf{ refuses } A \quad \text{implies} \quad S \textbf{ after } \sigma \textbf{ refuses } A \\ & \text{iff} & \forall \sigma \in L^*, \forall A \subseteq L : \\ & & \quad S \textbf{ after } \sigma \textbf{ must } A \quad \text{implies} \quad I \textbf{ after } \sigma \textbf{ must } A \end{array}$$

□

The relation $\leq_{te}$ is an interesting implementation relation. It is deduced naturally from testing implementations using the above argumentation, it is easily related to $\approx_{te}$, it is a preorder, and it implies $\leq_{tr}$ (proposition 3.10).

Proposition 3.10

1. $\approx_{te} \; = \; \leq_{te} \cap \leq_{te}^{-1}$
2. $\leq_{te}$ is a preorder.
3. $\leq_{te} \subseteq \leq_{tr}$

□

For conformance testing $\leq_{te}$ has a severe disadvantage: it is characterized using a quantification over all $\sigma \in L^*$, which poses the problem of having to verify by means of testing that S **after** σ **must** A implies I **after** σ **must** A for all $\sigma \in L^*$. In particular, this requires that it has to be verified that all traces *not* in specification S are also not in implementation I, as follows from the following characterization of $\leq_{tr}$:

Proposition 3.11
$I \leq_{tr} S$ iff $\forall \sigma \notin traces(S), \forall A \subseteq L : \; S \textbf{ after } \sigma \textbf{ must } A \quad \text{implies} \quad I \textbf{ after } \sigma \textbf{ must } A$

□

In [Bri88] the implementation relation *conformance* **conf** was introduced to reduce this problem (originally introduced as **imp** in [BSS87]). The relation **conf** reduces the quantification to traces in the specification, so that it does not test for $\leq_{tr}$. Taken together the relations $\leq_{tr}$ and **conf** exactly give $\leq_{te}$, as is expressed by proposition 3.13.1. This property suggests a way of *incremental testing*. In order to test for $\leq_{te}$, correctness with respect to $\leq_{tr}$ and **conf** can be separately checked, not necessarily using the same technique. We can think of checking **conf** by testing, while checking $\leq_{tr}$ by monitoring or verification.

Definition 3.12
Let $I, S \in \mathcal{LTS}$, then

$$\begin{array}{lll} I \textbf{ conf } S & =_{def} & \forall \sigma \in traces(S), \forall A \subseteq L : \\ & & \quad S \textbf{ after } \sigma \textbf{ must } A \quad \text{implies} \quad I \textbf{ after } \sigma \textbf{ must } A \end{array}$$

□

Proposition 3.13

1. $\leq_{te} \;=\; \leq_{tr} \cap \mathbf{conf}$
2. **conf** is reflexive, but not transitive.

□

Testing for **conf** means checking whether the implementation does not have unspecified deadlocks for traces in S. It is not checked whether an implementation has extra traces, i.e. extensions in the functionality of the implementation with respect to the specification remain undetected.

The relation **conf** can also be expressed using the observations Obs and Obs'.

Proposition 3.14

$$\begin{array}{lll} I \;\mathbf{conf}\; S & \text{iff} & \forall t \in \mathcal{LTS} : \;(\; Obs(t,I) \cap traces(S) \;) \;\subseteq Obs(t,S) \\ & \text{iff} & \forall t \in \mathcal{LTS} : \;(\; Obs(t,I) \cap traces(S) \;) \;\subseteq Obs(t,S) \;\text{ and} \\ & & \qquad\qquad\quad (\; Obs'(t,I) \cap traces(S) \;) \;\subseteq Obs'(t,S) \end{array}$$

□

Another characterization of **conf** can be given in terms of must tests by reintroducing the verdict **inconclusive**. As already noted in section 3.2 execution of a must test $t_{[\sigma,A]}$ either deadlocks while performing σ, or it deadlocks after σ has been performed refusing all actions in A, or it can reach a terminal state after having performed σ and one of the actions in A. If our interest is to know what happens *after* σ all deadlocks that occur *before* σ has been completely executed, are of no interest. If only such deadlocks can be observed we assign the verdict **inconclusive**. The verdict **pass** is assigned if σ can be observed and no deadlock occurs after σ, i.e. one of the actions in A can be performed. If I performs σ but refuses all actions in A, the verdict **fail** is assigned. The relation **conf** is obtained by requiring that all test cases that are successful, i.e. **pass**, with the specification are not unsuccessful, i.e. **pass** or **inconclusive**, with the implementation.

Proposition 3.15
Let $I, S \in \mathcal{LTS}$, and let application of a must test $t_{[\sigma,A]} \in \mathcal{LT}_M$ to I, S be defined by:

$$apply_{\mathcal{C}}(t_{[\sigma,A]}, I) \;=_{def}\; \begin{cases} \mathbf{pass} & \text{if } \sigma \notin Obs(t_{[\sigma,A]}, I) \\ & \quad \text{and } \exists a \in A : \sigma{\cdot}a \in Obs(t_{[\sigma,A]}, I) \\ \mathbf{inconclusive} & \text{if } \sigma \notin Obs(t_{[\sigma,A]}, I) \\ & \quad \text{and } \forall a \in A : \sigma{\cdot}a \notin Obs(t_{[\sigma,A]}, I) \\ \mathbf{fail} & \text{if } \sigma \in Obs(t_{[\sigma,A]}, I) \end{cases}$$

then

$$I \;\mathbf{conf}\; S \;\text{ iff }\; \forall t \in \mathcal{LT}_M : \quad apply_{\mathcal{C}}(t,S) = \mathbf{pass} \quad \text{implies} \quad apply_{\mathcal{C}}(t,I) \neq \mathbf{fail}$$

□

In the chapters 4 and 6 the implementation relation **conf** will be used as the most important implementation relation for conformance testing. The relation $\leq_{te}$ is more of theoretical interest, e.g. as the basis for correctness verification or correctness preserving transformations (section 1.1). Note that an implementation that is tested and found non-conforming according to **conf** is also not correct according to $\leq_{te}$, but that the converse does not hold.

3.4 The Conformance Relation CONF

Since the implementation relation **conf** is used in the next chapters as our notion of conformance for conformance testing, this section elaborates on **conf**. **conf** is studied within the testing framework of chapter 2. A definition of passing a test for labelled transition system tests is given, and a new class of deterministic tests is introduced. Finally, we consider logical **conf**-theories.

3.4.1 Requirements for CONF

It is easily checked, using proposition 3.12, that **conf** can be defined as an implementation relation compatible according to definition 2.10 with the requirement language $\mathcal{L}_{must}$, together with the relations $\mathbf{spec}_{\mathcal{C}}$ and $\mathbf{sat}_{\mathcal{C}}$.

Definition 3.16
Let $r =$ **after** σ **must** $A \in \mathcal{L}_{must}$, then $\mathbf{spec}_{\mathcal{C}}, \mathbf{sat}_{\mathcal{C}} \subseteq \mathcal{LTS} \times \mathcal{L}_{must}$ are defined by

- S $\mathbf{spec}_{\mathcal{C}}$ r $=_{def}$ $S \stackrel{\sigma}{\Longrightarrow}$ and S **after** σ **must** A
- I $\mathbf{sat}_{\mathcal{C}}$ r $=_{def}$ I **after** σ **must** A

The requirements specified by a specification S according to $\mathbf{spec}_{\mathcal{C}}$, respectively satisfied by an implementation I according to $\mathbf{sat}_{\mathcal{C}}$, are denoted by:

- $specs_{\mathcal{C}}(S)$ $=_{def}$ $\{\ r \in \mathcal{L}_{must} \mid S\ \mathbf{spec}_{\mathcal{C}}\ r\ \}$
- $sats_{\mathcal{C}}(I)$ $=_{def}$ $\{\ r \in \mathcal{L}_{must} \mid I\ \mathbf{sat}_{\mathcal{C}}\ r\ \}$

□

Proposition 3.17
The implementation relation **conf** is compatible with the requirement language $\mathcal{L}_{must}$, specification relation $\mathbf{spec}_{\mathcal{C}}$, and satisfaction relation $\mathbf{sat}_{\mathcal{C}}$:

$$I\ \mathbf{conf}\ S \quad \text{iff} \quad I\ \mathbf{sat}_{\mathcal{C}}\ specs_{\mathcal{C}}(S)$$

□

We have $\mathbf{spec}_{\mathcal{C}} \subset \mathbf{sat}_{\mathcal{C}}$, so (proposition 2.3) **conf** is reflexive. We already noticed that **conf** is not transitive (proposition 3.13.2).

Remark 3.18
Note that intransitivity cannot be concluded from proposition 2.4. The first requirement is not fulfilled: let $I \in \mathcal{LTS}$, then there is no $S \in \mathcal{LTS}$, such that $specs_{\mathcal{C}}(S) = sats_{\mathcal{C}}(I)$, viz. take I and $\sigma \in L^*$, such that $\sigma \notin traces(I)$, implying I **after** σ **must** $\emptyset \in sats_{\mathcal{C}}(I)$. Then S must specify **after** σ **must** $\emptyset$, which is not possible: S $\mathbf{spec}_{\mathcal{C}}$ **after** σ **must** $\emptyset$ iff $\sigma \in traces(S)$ and $\sigma \notin traces(S)$, hence such an S does not exist. □

Example 3.19
Consider again the vending machine in figure 2.7 as a behaviour specification B_M. The set of conformance requirements specified by B_M for **conf** is given by

$$\begin{aligned} specs_{\mathcal{C}}(B_M) &= \{ \quad r \in \mathcal{L}_{must} \mid B_M \ \mathbf{spec}_{\mathcal{C}} \ r \ \} \\ &= \{ \quad \mathbf{after}\ \epsilon\ \mathbf{must}\ \{shilling\}\ ,\ \ \mathbf{after}\ \epsilon\ \mathbf{must}\ \{shilling, coffee\}\ , \ldots , \\ &\qquad \mathbf{after}\ shilling\ \mathbf{must}\ \{coffee\text{-}button\}\ , \\ &\qquad \mathbf{after}\ shilling\ \mathbf{must}\ \{tea\text{-}button\}\ , \\ &\qquad \mathbf{after}\ shilling\ \mathbf{must}\ \{coffee\text{-}button,\ tea\text{-}button\}\ , \ldots , \\ &\qquad \mathbf{after}\ shilling{\cdot}coffee\ \mathbf{must}\ \{coffee\text{-}button\}\ , \ldots , \\ &\qquad \mathbf{after}\ shilling{\cdot}tea\ \mathbf{must}\ \{tea\text{-}button\}\ , \ldots \ \} \end{aligned}$$

Consider the potential implementations in figure 2.8: whereas only $I_1 \leq_{te} B_M$, we now have: I_1, I_2, I_3 **conf** B_M. The relation **conf** does allow extra traces in the implementation, which are not in the specification, e.g. the trace *penny* of I_2. I_2 does not satisfy the requirement **after** *penny* **must** $\emptyset$ of S_{must} (cf. section 2.4.2), but this is not a specified requirement for **conf**, since $B_M \stackrel{penny}{\Longrightarrow\!\!\!\!\!\!\not\ }$.

However, branches cannot be added arbitrarily: in the nondeterministic specification of figure 2.10 supplying *tea* is optional: I_1, I_5 **conf** figure 2.10. But adding a branch *tea-button* in the implementation I_5 is not allowed: I_7 **co$\not$nf** B_M. Once the action *tea-button* has been accepted also *tea* must be produced; I_7 does not satisfy the requirement **after** *shilling*$\cdot$*tea-button* **must** $\{tea\}$.

□

3.4.2 Nondeterministic Tests with Trace Based Verdicts

In the previous section implementation relations were introduced based on observations that test cases can make of implementations. For conformance testing we need to know which observations make a test successful, i.e. we need to evaluate the observations (cf. section 2.2.4), so that we can define the relation **passes** $\subseteq \mathcal{LTS} \times \mathcal{L}_T$ (section 2.5).

In this subsection test cases in $\mathcal{LTS}$ are considered, with a corresponding relation $\mathbf{passes}_{\mathcal{N}} \subseteq \mathcal{LTS} \times \mathcal{LTS}$. For $\mathbf{passes}_{\mathcal{N}}$ the deadlock observations in $Obs(t, I)$ are evaluated using the deadlocks of t: all deadlocks in $Obs(t, I)$ must also be deadlocks of t.

Definition 3.20
Let $t, I \in \mathcal{LTS}$, then $I\ \mathbf{passes}_{\mathcal{N}}\ t \ =_{def}\ \forall \sigma \in Obs(t, I) :\ t\ \mathbf{after}\ \sigma\ \mathbf{deadlocks}$

□

Remark 3.21
Note that $t\ \mathbf{after}\ \sigma\ \mathbf{deadlocks}$ means that t must have the possibility of reaching deadlock after performing σ, not that the resulting state of t in a particular test run is a deadlock. We come back to this in the next subsection.

□

For a test case $t \in \mathcal{LTS}$ the tested requirements (section 2.5) are expressed in proposition 3.22 by $testreqs_{\mathcal{N}}(t)$. These requirements are satisfied if and only if the test is passed (cf. (2.32)).

Proposition 3.22
Let $testreqs_{\mathcal{N}} : \mathcal{LTS} \rightarrow \mathcal{P}(\mathcal{L}_{must})$ be defined by

$$testreqs_{\mathcal{N}}(t) \;=_{def}\; \{\, \textbf{after } \sigma \textbf{ must } A \;\mid\; t \textbf{ after } \sigma \textbf{ must } L \;\text{ and } \exists t' \,(\, t \stackrel{\sigma}{\Longrightarrow} t' \;\text{ and }\; out(t') \subseteq A \,) \,\}$$

then

$$I \textbf{ passes}_{\mathcal{N}}\ t \quad \text{iff} \quad I \textbf{ sat}_{\mathcal{C}}\ testreqs_{\mathcal{N}}(t)$$

□

Proposition 3.23
Let $S \in \mathcal{LTS}$, then the test suite

$$\Pi^{tr}_{\textbf{conf}}(S) \;=_{def}\; \{\, t \in \mathcal{LTS} \mid traces(t) \subseteq traces(S),\ S \textbf{ passes}_{\mathcal{N}}\ t \,\}$$

is complete for **conf**-conforming implementations of S, i.e.

$$\forall I \in \mathcal{LTS} : \;\; I \textbf{ conf } S \quad \text{iff} \quad I \textbf{ passes}_{\mathcal{N}}\ \Pi^{tr}_{\textbf{conf}}(S)$$

□

The test suite $\Pi^{tr}_{\textbf{conf}}(S)$ is complete: it detects all and only incorrect implementations. For applying it to practical conformance testing it has too many superfluous test cases. This means that selection of a suitable subset Π' of test cases is necessary, preferably such that the testing power is not affected, i.e. Π' should also be complete. In chapter 4 test generation methods that algorithmically generate test suites from S that are more efficient than $\Pi^{tr}_{\textbf{conf}}(S)$ are studied.

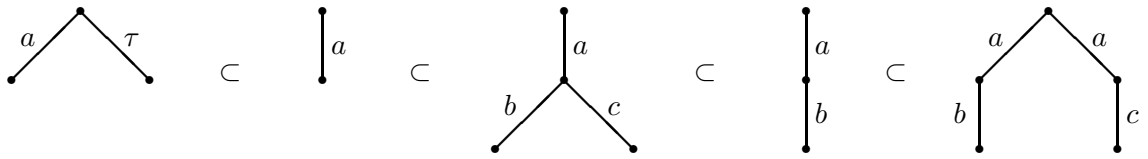


Figure 3.2: Testing power of test cases.

Example 3.24
Test cases can be related to each other based on their testing power, i.e. the requirements $testreqs_{\mathcal{N}}$ that they test. A test case is more powerful if it tests more requirements. In figure 3.2 a few test cases are related with respect to their testing power. The testing power increases from left to right: the left-most test case does not test any requirement, the next one tests **after** ϵ **must** $\{a\}$, the third one also tests **after** a **must** $\{b, c\}$,

the fourth one adds **after** a **must** $\{b\}$ to the tested requirements, and the fifth test case tests all these requirements plus **after** a **must** $\{c\}$.

It can be noted that a relation on $\mathcal{LTS}$ based on testing power is not easily related to one of the previously defined relations on $\mathcal{LTS}$.

□

3.4.3 Deterministic Tests with State Based Verdicts

We now consider test cases that have states labelled with the verdicts **pass** or **fail**. Again test execution consists of trying to perform as many actions as possible, but when a non-terminal state of the test case is reached this does not automatically imply a **fail**-verdict; the verdict assigned is the label of that state (cf. remark 3.21). We introduce a special kind of deterministic labelled transition systems $\mathcal{DLTS}$ to model these tests.

Definition 3.25

1. A *state labelled test case* in $\mathcal{DLTS}$ is a 5-tuple $\langle S, L, T, s_0, v\rangle$, such that $\langle S, L, T, s_0\rangle$ is a deterministic labelled transition system, and $v : S \to \{\mathbf{pass}, \mathbf{fail}\}$ is a verdict function.

   Definitions applicable to $\mathcal{LTS}$ are extended to $\mathcal{DLTS}$ by defining them over the underlying labelled transition system.

2. Let $I \in \mathcal{LTS}$, $t \in \mathcal{DLTS}$, then

$$I \ \mathbf{passes}_{\mathcal{D}} \ t \quad =_{def} \quad \forall \sigma \in \mathit{Obs}(t, I) : v(\, t \ \mathbf{after} \ \sigma \,) = \mathbf{pass}$$

□

Note that $v(\, t \ \mathbf{after} \ \sigma \,)$ is well-defined: $\sigma \in \mathit{Obs}(t, I)$ implies $\sigma \in \mathit{traces}(S)$, so there exists a state $t \ \mathbf{after} \ \sigma$, and determinism of t guarantees that there is at most one such a state.

Analogous to the previous subsection the tested requirements can be determined, and a complete test suite of state labelled test cases can be derived.

Proposition 3.26
Let $\mathit{testreqs}_{\mathcal{D}} : \mathcal{DLTS} \to \mathcal{P}(\mathcal{L}_{must})$ be defined by

$$\mathit{testreqs}_{\mathcal{D}}(t) \quad =_{def} \quad \{\, \mathbf{after} \ \sigma \ \mathbf{must} \ A \ \mid \ v(\, t \ \mathbf{after} \ \sigma \,) = \mathbf{fail} \ \text{and} \ \mathit{out}(\, t \ \mathbf{after} \ \sigma \,) \subseteq A \,\}$$

then

$$I \ \mathbf{passes}_{\mathcal{D}} \ t \quad \text{iff} \quad I \ \mathbf{sat}_{\mathcal{C}} \ \mathit{testreqs}_{\mathcal{D}}(t)$$

□

Proposition 3.27
Let $S \in \mathcal{LTS}$, then the test suite

- $\Pi^{det}_{\mathbf{conf}}(S) \;=_{def}\; \{\; t \in \mathcal{DLTS} \mid traces(t) \subseteq traces(S),\; S\ \mathbf{passes}_{\mathcal{D}}\ t \;\}$

is complete for **conf**-conforming implementations of S.

□

Remark 3.28
We did not include the verdict **inconclusive** in state labelled test cases. Sometimes we will use it in the next chapters to indicate a test outcome which is not **fail**, but also not intended. In such cases I $\mathbf{passes}_{\mathcal{D}}$ t should be read as $v(\, t\, \mathbf{after}\, \sigma\,) \neq \mathbf{fail}$, so that semantically the verdict **inconclusive** is the same as the verdict **pass**: no evidence of non-conformance was found, that justifies assigning the verdict **fail**. (See also the introduction of the relation **passes** in section 2.5).

□

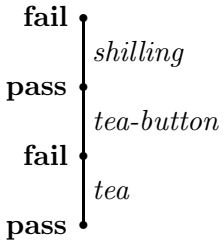


Figure 3.3: Example test case $t \in \mathcal{DLTS}$.

Example 3.29
Figure 3.3 specifies a test case in $\mathcal{DLTS}$ that can be used to test the nondeterministic vending machine of figure 1.7 (section 1.4). Test case t specifies to insert a *shilling*, followed by pushing the *tea-button*, after which *tea* must be obtained. If the machine under test indeed accepts the *shilling*, unlocks the *tea-button*, and supplies *tea* the verdict **pass** is assigned. If the *tea-button* is not unlocked the verdict is also **pass**, but if the purpose of the test case is to test the *tea-button*·*tea* branch, its intuitive meaning is **inconclusive**. If the *tea-button* can be pushed but no *tea* is obtained the test **fails**.

Test case t tests for the requirements **after** ϵ **must** $\{$*shilling*$\}$, and **after** *shilling*·*tea-button* **must** $\{$*tea*$\}$.

□

For **conf**-test suites test cases in $\mathcal{LTS}$ with passing expressed by $\mathbf{passes}_{\mathcal{N}}$, and test cases in $\mathcal{DLTS}$ with $\mathbf{passes}_{\mathcal{D}}$, can be expressed in each other: a state labelled test case is transformed into a test case in $\mathcal{LTS}$ by adding $\mathbf{i}; \mathbf{stop}$ branches in all non-terminal states with verdict **pass**. A test case in $\mathcal{LTS}$ is expressed by a test suite in $\mathcal{DLTS}$. The details of these transformations are not elaborated here. In the next chapters those test cases are used that best suit the relevant needs.

Note that the combination of deterministic test cases in $\mathcal{DLTS}$ and $\mathbf{passes}_{\mathcal{N}}$, which corresponds to having **pass** verdicts in terminal states only, is not powerful enough for **conf**-testing. The requirement **after** $a \cdot b$ **must** $\{c\}$ of S in figure 3.4 cannot be tested with a test case that is deterministic and does not have the verdict **pass** in a

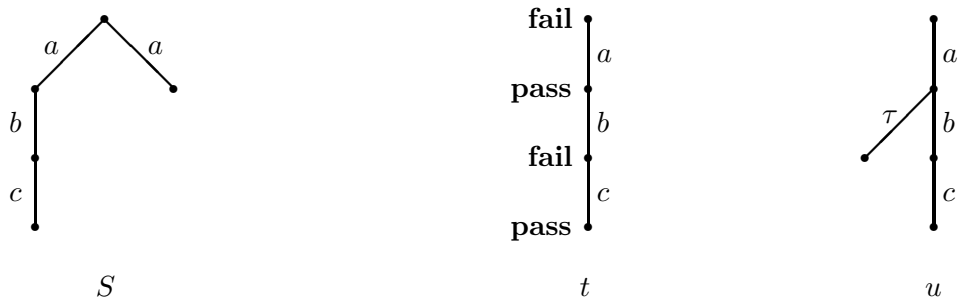


Figure 3.4: Testing with $\mathcal{DLTS}$ and $\mathbf{passes}_{\mathcal{N}}$.

non-terminal state. To take into account the possible deadlock after a either the state after a must be labelled **pass** as in t, or there must be the possibility that the test case deadlocks after a as in u.

3.4.4 CONF and Logic

The relation **conf** can be studied in a logical framework, like $\mathcal{L}_{\overline{tr}}$ in section 2.4.4. To illustrate this we study some logical concepts for **conf**-theories, i.e. sets of requirements $T \subseteq \mathit{specs}_{\mathcal{C}}(S)$ for some $S \in \mathcal{LTS}$, over the class of models $\mathcal{LTS}$. We start with the concept of derivation.

For $\mathcal{L}_{must}$ there are no axioms, i.e. there are no formulae in $\mathcal{L}_{must}$ that hold for any process. This can be seen as follows: suppose $\mathbf{after}\ \sigma\ \mathbf{must}\ A$ would be such an axiom, with $\sigma = b_1 \cdot b_2 \cdot \ldots \cdot b_m \in L^*$, then $b_1; b_2; \ldots; b_m; \mathbf{stop}\ \not\mathbf{sat}_{\mathcal{C}}\ \mathbf{after}\ \sigma\ \mathbf{must}\ A$.

Inference rules for **conf**-theories are inspired by proposition 3.5.3. Let $A_1 \subseteq A_2$, then

$$\mathbf{after}\ \sigma\ \mathbf{must}\ A_1 \ \vdash\ \ \mathbf{after}\ \sigma\ \mathbf{must}\ A_2 \tag{3.4}$$

Derivation of a requirement $r \in \mathcal{L}_{must}$ from a **conf**-theory T, $S \vdash r$, is described for the general case in section 2.4.4. For the inference rule (3.4) this can be simplified:

- the inference rule has only one premiss, hence derivations are *linear*;
- a sequence of applications of (3.4) can be replaced by one application of (3.4).

Definition 3.30
Derivation for a **conf**-theory $T \subseteq \mathit{specs}_{\mathcal{C}}(S)$, for some $S \in \mathcal{LTS}$, is defined by

$$T \vdash\ \mathbf{after}\ \sigma\ \mathbf{must}\ A \quad =_{def} \quad \exists A' \subseteq A :\ \ \mathbf{after}\ \sigma\ \mathbf{must}\ A'\ \in T$$

□

Proposition 3.31
Derivation for **conf**-theories T over the class of models $\mathcal{LTS}$ with the satisfaction relation $\mathbf{sat}_{\mathcal{C}}$ is sound and complete, i.e.

$$T \vdash r \quad \text{iff} \quad \forall I \in \mathcal{LTS} : I \ \mathbf{sat}_{\mathcal{C}} \ T \ \text{ implies } \ I \ \mathbf{sat}_{\mathcal{C}} \ r$$

□

A set of requirements is logically independent if no requirements can be removed without changing its meaning (section 2.4.4, equations (2.27) and (2.28)). To obtain an independent **conf**-theory we must remove all requirements that can be derived from other requirements in T, i.e. a requirement $\mathbf{after}\ \sigma\ \mathbf{must}\ A$ must be removed if there is a requirement $\mathbf{after}\ \sigma\ \mathbf{must}\ A' \in S$ with $A' \subset A$.

Proposition 3.32
A **conf**-theory T is logically independent if and only if $\forall \sigma \in L^* : M_\sigma = min_{\subseteq}(M_\sigma)$, where $M_\sigma =_{def} \{\ A \subseteq L \mid \mathbf{after}\ \sigma\ \mathbf{must}\ A \in T\ \}$.

□

Independent sets of requirements can be used as minimal representations of sets of requirements. Unfortunately, like for $\mathcal{L}_{\overline{tr}}$ (section 2.4.4), independent **conf**-theories cannot always be obtained.

Example 3.33
For S no independent **conf**-theory exists (figure 3.5):

$$S \ = \ \Sigma\ \{\ \mathbf{i};\ \Sigma\ \{\ a_i; \mathbf{stop} \mid i \geq n\ \} \mid n \in \mathbf{N}\ \}$$

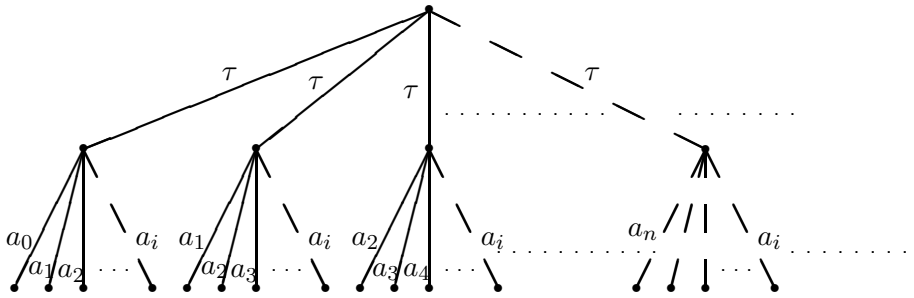


Figure 3.5: S without independent **conf**-theory.

$$\begin{aligned} specs_{\mathcal{C}}(S) \ \supseteq \ \{ \quad & \mathbf{after}\ \epsilon\ \mathbf{must}\ \{a_0, a_1, a_2, a_3, a_4, \ldots\}\ , \\ & \mathbf{after}\ \epsilon\ \mathbf{must}\ \{a_1, a_2, a_3, a_4, \ldots\}\ , \\ & \mathbf{after}\ \epsilon\ \mathbf{must}\ \{a_2, a_3, a_4, \ldots\}\ , \\ & \mathbf{after}\ \epsilon\ \mathbf{must}\ \{a_3, a_4, \ldots\}\ , \ldots\} \\ = \ \{ \quad & \mathbf{after}\ \epsilon\ \mathbf{must}\ \{\ a_i \mid i \geq n\ \} \mid n \in \mathbf{N}\ \ \} \end{aligned}$$

Viewing $\vdash$ as a relation on $\mathcal{L}_{must}$ this corresponds to $\langle \mathcal{L}_{must}, \vdash \rangle$ not being well-founded (appendix A), which results from the non-well-foundedness of $\langle \mathcal{P}(L), \subseteq \rangle$. We can make derivations that are infinite 'to the left':

$$\begin{array}{ll} & \vdots \\ \vdash & \textbf{after } \epsilon \textbf{ must } \{a_3, a_4, \ldots\} \\ \vdash & \textbf{after } \epsilon \textbf{ must } \{a_2, a_3, a_4, \ldots\} \\ \vdash & \textbf{after } \epsilon \textbf{ must } \{a_1, a_2, a_3, a_4, \ldots\} \\ \vdash & \textbf{after } \epsilon \textbf{ must } \{a_0, a_1, a_2, a_3, a_4, \ldots\} \end{array}$$

□

The process I_L that can always execute any action in L, satisfies any **conf**-theory, implying that any **conf**-theory is consistent:

$$I_L \quad =_{def} \quad \Sigma\{\ a; I_L \mid a \in L\ \}$$

Remark 3.34
Elaborating these logical concepts for arbitrary theories in $\mathcal{L}_{must}$ also containing requirements of the form $\textbf{after } \sigma \textbf{ must } \emptyset$ is feasible, but more complicated. In particular, derivation cannot be defined in a simple way like in definition 3.30, implying that the general definition in section 2.4.4 must be used.

□

3.5 Concluding Remarks

This chapter introduced testing equivalence $\approx_{te}$ as a natural equivalence for distributed systems. The implementation relations $\leq_{te}$ and **conf** were introduced to serve as implementation relations. The relation $\leq_{te}$ is theoretically the most interesting. However, for the purpose of conformance testing it has the disadvantage of being defined using a quantification over all $\sigma \in L^*$, which poses the problem of also having to test properties for traces not in the specification. The implementation relation **conf** is weaker: only properties of traces in the specification have to be verified. Consequently, a **conf**-conforming implementation may exhibit extra functionality with respect to the specification. In the next chapter we will consider **conf** as the main relation on which test generation will be based.

The relations $\leq_{te}$ and **conf** are not the only possible implementation relations for labelled transition systems. Many others have been defined in literature. Examples are the extension relation **ext** in [BS86], variations on **conf** in [Led90], the relations described in [Gla90]. Variations in implementation relations can for instance be obtained by varying the testers [Phi87, Abr87, Lan90], or varying the way test cases are applied to implementations, in particular leaving the assumption of synchronous communication between test case and implementation. This last possibility is elaborated in chapter 5.

The existence of many implementation relations raises the question which one to use for a particular application. In chapter 7 this question is discussed as one of the open problems.

Chapter 4

Synchronous Testing

4.1 Introduction

In the realm of labelled transition systems with synchronous observations the implementation relation **conf**, defined and studied in chapter 3, is a reasonable candidate to formalize the notion of conformance for the purpose of conformance testing. Therefore a natural next question is how to derive test suites for **conf** systematically from a labelled transition system specification S. This section investigates such methods for test derivation. An important point for these methods is that they should lead to implementable algorithms, in order to be applicable to test derivation from realistically sized specifications.

In the next section a test derivation algorithm is developed, starting from the fact that for a complete test suite the requirements specified by a specification should be exactly those tested by the derived test suite. The derived test cases are deterministic with state based verdicts ($\mathcal{DLTS}$, section 3.4.3). All derived test cases are then combined into one nondeterministic test case ($\mathcal{LTS}$, section 3.4.2): the canonical tester of [Bri87, Bri88].

If a specification is given by a behaviour expression with labelled transition system semantics, there are two possibilities for the derivation of test cases. Either the behaviour expression is replaced by its semantics, from which tests are derived using algorithms for labelled transition systems, or the algorithms are transformed to work on behaviour expressions as well. The first option poses problems when language constructs are used that represent infinite labelled transition systems: finite, implementable representations for infinite labelled transition systems have to be found. Of course, behaviour expressions themselves are a logical candidate for this representation, leading to the second option for the derivation of tests from behaviour expressions. In section 4.3 this method is explored, using a subclass of behaviour expressions for which the semantics is always a finite labelled transition system. In section 4.4 the language is extended to represent infinite labelled transition systems.

Test derivation for **conf** from labelled transition systems is studied in [BSS87, Bri87, Bri88]. Section 4.2 is based on them. In [Eer87] an implementation of test derivation

for a subset of the language LOTOS [BB87, ISO89b] is described based on the first option above: language expressions are transformed to labelled transition systems from which tests are derived. The CO-OP method [Wez90] is an example of the second option: tests are derived by deriving attributes, viz. *Compulsory* and *Options*, directly from behaviour expressions. The CO-OP method also works for a subset of LOTOS, however, with this subset no infinite labelled transition systems can be represented. An implementation is described in [Ald90]. First attempts to cope with infinite labelled transition systems have been made in [Tre90, Doo91].

4.2 Test Derivation for Labelled Transition Systems

To do systematic test derivation from a labelled transition system S an algorithm is required to construct test cases that are sound, and preferably, when taken together, also exhaustive. This means that test cases must be derived, such that the requirements tested by these test cases are exactly the requirements specified by S (equation (2.33)). For a **conf**-test suite Π in $\mathcal{DLTS}$ this means that (definition 3.16, proposition 3.26):

$$\mathit{testreqs}_{\mathcal{D}}(\Pi) \;=\; \mathit{specs}_{\mathcal{C}}(S) \tag{4.1}$$

Hence exactly for every $\mathbf{after}\ \sigma\ \mathbf{must}\ A \in \mathit{specs}_{\mathcal{C}}(S)$, i.e. for every σ, A with

$$S \stackrel{\sigma}{\Longrightarrow} \quad \text{and} \quad S\ \mathbf{after}\ \sigma\ \mathbf{must}\ A \tag{4.2}$$

there must be $\mathbf{after}\ \sigma\ \mathbf{must}\ A \in \mathit{testreqs}_{\mathcal{D}}(\Pi)$, i.e. there must be a test case $t \in \Pi$, with

$$v(\, t\ \mathbf{after}\ \sigma\,) = \mathbf{fail} \quad \text{and} \quad \mathit{out}(\, t\ \mathbf{after}\ \sigma\,) \subseteq A \tag{4.3}$$

Moreover, t should not test requirements not in $\mathit{specs}_{\mathcal{C}}(S)$.

Proposition 3.27 already presented such a test suite, but the test suite $\Pi^{det}_{\mathbf{conf}}(S)$ contains redundant test cases, and it is not systematically derived from S in such a way that it could easily be implemented in a test derivation algorithm. We will now derive an algorithm with which test cases can be generated from a labelled transition system specification S. The algorithm is derived in three steps: first it is assumed that the set of observable actions L is finite, and that S has finite behaviour. This results in the algorithm in proposition 4.5. In the second step we give up the assumption of finite behaviour, and in the final step also L may be infinite. The resulting algorithm is presented in proposition 4.10.

Finite L, finite behaviour

The first step in generating the test cases is the derivation from S of the requirements $\mathbf{after}\ \sigma\ \mathbf{must}\ A$ according to (4.2).

We can start with $\sigma = \epsilon$, and determine all $A \subseteq L$ with S **after** ϵ **must** A . Let $A \subseteq L$ with **after** ϵ **must** A , then for this A there must be a test case t_A such that (4.3):

$$v(\, t_A \,\mathbf{after}\, \epsilon\,) = v(t_A) = \mathbf{fail} \quad \text{and} \quad out(\, t_A \,\mathbf{after}\, \epsilon\,) = out(t_A) \;\subseteq\; A$$

Moreover, t_A should not test requirements not specified by S. This is accomplished by having $out(t_A) = A$:

$$t_A \;=\; \Sigma\{a; t_a \mid a \in A\} \;\text{ and }\; v(t_A) = \mathbf{fail}$$

where t_a is the behaviour of the test case after a, which is elaborated below.

The test case t_A tests not only for requirement **after** ϵ **must** A , but also for all requirements **after** ϵ **must** A' with $A \subseteq A'$. But since S **after** ϵ **must** A implies **after** ϵ **must** A' for all $A' \supseteq A$ (proposition 3.5.3), no unspecified requirements are tested. This property implies that if there is a test case t_A that tests requirement **after** ϵ **must** A , all test cases $t_{A'}$ with $A \subseteq A'$ can be removed. Hence, it suffices to derive test cases for those sets A satisfying S **after** ϵ **must** A , that are minimal with respect to $\subseteq$.

Since for each A satisfying S **after** ϵ **must** A always $A \cap out(S) \subseteq A$, and S **after** ϵ **must** $(A \cap out(S))$, it suffices to consider only $A \subseteq out(S)$. This leads to the definition of the *must set* $\overline{M_\epsilon}(S)$. Test cases t_A must be made for each $A \in min_\subseteq(\overline{M_\epsilon}(S))$.

Definition 4.1
Let $S \in \mathcal{LTS}$, then $\overline{M_\epsilon}(S) \;=_{def}\; \{\; A \subseteq out(S) \mid S \,\mathbf{after}\, \epsilon \,\mathbf{must}\, A \,\}$ □

The next step is to construct the behaviour of the test case after a: t_a. t_a must test the requirements **after** a **must** A . We can make this a recursive procedure if we can repeat the calculation with $\sigma = \epsilon$, i.e. if there is an S' such that

$$S \,\mathbf{after}\, a \,\mathbf{must}\, A \quad \text{iff} \quad S' \,\mathbf{after}\, \epsilon \,\mathbf{must}\, A \tag{4.4}$$

Such S' can be defined; it is given by the expression **choice** S **after** a (definition 4.2), and it indeed satisfies property (4.4) (proposition 4.3.3, using that $a \in out(S) \subseteq traces(S)$).

The more general expression **choice** S **after** σ defines the nondeterministic choice among all states that can be reached by S after having performed σ. It models exactly the behaviour of S after σ up to $\approx_{te}$, as is expressed by propositions 4.3.2 and 4.3.3.

Definition 4.2
Let $S \in \mathcal{LTS}$, $\sigma \in L^*$, then **choice** S **after** $\sigma \;=_{def}\; \Sigma\{\; \mathbf{i}; S' \mid S' \in S \,\mathbf{after}\, \sigma \,\}$ □

Proposition 4.3
Let $S \in \mathcal{LTS}$, $\sigma, \sigma_1, \sigma_2 \in L^*$, $A \subseteq L$, then

1. **choice** S **after** σ $=_{def}$ $\Sigma\{\ \mathbf{i}; S' \mid S \stackrel{\sigma}{\Longrightarrow} S'\ \}$
   $\approx_{te}$ $\Sigma\{\ \mathbf{i}; S' \mid S \stackrel{\sigma}{\Longrightarrow} S'\ \}$
2. **choice** S **after** ϵ $\approx_{te}$ S
3. For $\sigma_1 \in traces(S)$ or $\sigma_2 \neq \epsilon$:

$$S \textbf{ after } \sigma_1 \cdot \sigma_2 \textbf{ must } A \quad \text{iff} \quad (\textbf{ choice } S \textbf{ after } \sigma_1) \textbf{ after } \sigma_2 \textbf{ must } A$$

4. **choice** S **after** $\sigma_1 \cdot \sigma_2$ $\approx_{te}$ **choice** (**choice** S **after** σ_1) **after** σ_2

□

Now construction of t_a consists of applying recursively the previous steps to the specification **choice** S **after** a . In order to test all requirements **after** a **must** A for all $a \in out(S)$, we must be sure that for each $a \in out(S)$ there is at least one test case t_A that can make an a-transition. Due to the minimalization in the previous step this is not guaranteed, see example 4.4.

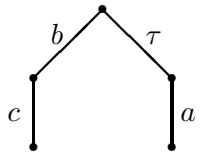


Figure 4.1:

Example 4.4
In figure 4.1 $\overline{M_\epsilon}(S) = \{\{a\}, \{a, b\}\}$, and $min_\subseteq(\overline{M_\epsilon}(S)) = \{\{a\}\}$. If only test cases are made for $min_\subseteq(\overline{M_\epsilon}(S))$, then there is no test case to make a b-transition: the requirement **after** b **must** $\{c\}$ cannot be tested.

□

Guaranteeing transitions for all $a \in out(S)$ can be accomplished in different ways by adding test cases with verdict **pass**: any test case $t \in \mathcal{DLTS}$ with $v(\, t \textbf{ after } \epsilon\,) = v(t) =$ **pass** can always be added to a test suite, without affecting the soundness. In this case we can add t_A with

$$A = out(S) \backslash \bigcup min_\subseteq(\overline{M_\epsilon}(S)) \quad \text{or} \quad A = out(S)$$

Another possibility is to add test cases with verdict **fail**, such that they do not test additional requirements. This can be accomplished by choosing any set $M \supseteq min(\overline{M_\epsilon}(S))$, such that no labels 'are lost':

$$\bigcup \overline{M_\epsilon}(S) = \bigcup M \tag{4.5}$$

For the moment we take the first possibility, and add the test case $t_{out(S)}$.

The above argumentation leads to proposition 4.5. Proposition 4.5.1 presents an algorithm to obtain a sound test case; proposition 4.5.2 gives sufficient sound test cases

for a complete test suite. The recursion is guaranteed to terminate by the finite behaviour that is assumed for S: after a finite number of recursions there is S' such that $out(S') = \emptyset$, hence $A \subseteq out(S')$ implies $A = \emptyset$, and $t_\emptyset = \Sigma\emptyset = \mathbf{stop}$.

Proposition 4.5
Let L be finite, and let $S \in \mathcal{LTS}$ have finite behaviour.

1. $t_A \;=_{def}\; \Sigma\{a; t_a \mid a \in A\}$ is a sound test case for S, if
   $A \subseteq out(S)$, and
   $v(t_A) = \mathbf{fail}$ implies S **after** ϵ **must** A, and
   t_a is a sound test case for **choice** S **after** a.
2. The test suite

$$\Pi^1_{\mathbf{conf}}(S) \;=_{def}\; \{\; t_A \in \mathcal{DLTS} \mid\; t_A = \Sigma\{a; t_a \mid a \in A\},\; (\; A \in min_{\subseteq}(\overline{M_\epsilon}(S)) \text{ and } v(t_A) = \mathbf{fail} \;) \text{ or } (\; A = out(S) \text{ and } v(t_A) = \mathbf{pass} \;),\; t_a \in \Pi^1_{\mathbf{conf}}(\; \mathbf{choice}\; S\; \mathbf{after}\; a \;)\; \},$$

   is complete.

□

Finite L, infinite behaviour

If the behaviour of S is infinite this influences proposition 4.5 in the sense that the recursion is not guaranteed to terminate. It can be terminated at any moment by choosing $A = \emptyset$, which makes $t_\emptyset = \Sigma\emptyset = \mathbf{stop}$. The verdict $v(t_\emptyset)$ must be **pass**. The choice $A = \emptyset$ with the corresponding verdict **pass** is always valid: it does not test for any requirement, hence it does not affect soundness. By making $A = \emptyset$ when $out(S) \neq \emptyset$, finite test cases can be obtained also for specifications S with infinite behaviour. A complete test suite is obtained by combining all test cases with finite, but arbitrarily long behaviour. This principle of describing infinite behaviour by a set of processes with arbitrary, finite length is referred as the *approximation induction principle* [Bae86].

Infinite L, infinite behaviour

If L is infinite a problem arises for $min_{\subseteq}(\overline{M_\epsilon}(S))$ in proposition 4.5.2. Because of non-well-foundedness of the poset $\langle \mathcal{P}(L), \subseteq \rangle$ for infinite L, minimal elements in $\overline{M_\epsilon}(S)$ need not exist.

Example 4.6
Consider again S in figure 3.5, example 3.33. There is the following infinite sequence

of sets $A \subseteq L$, each satisfying S **after** ϵ **must** A :

$$\begin{array}{ll} & \vdots \\ \subseteq & \{a_3, a_4, \ldots\} \\ \subseteq & \{a_2, a_3, a_4, \ldots\} \\ \subseteq & \{a_1, a_2, a_3, a_4, \ldots\} \\ \subseteq & \{a_0, a_1, a_2, a_3, a_4, \ldots\} \end{array}$$

There is no minimal element in $\overline{M_\epsilon}(S)$.

□

A sufficient condition for minimal elements to exist is image-finiteness of S (definition 1.11.6): if S **after** ϵ is finite, then for each A with S **after** ϵ **must** A there exists a finite $A' \subseteq A$ with S **after** ϵ **must** A' , viz. $A' = \{a_1, a_2, \ldots, a_n\}$ with $a_i \in out(S_i)$ for each $S_i \in S$ **after** ϵ . If $\not\exists a_i \in out(S_i)$ for some i then $\overline{M_\epsilon}(S) = \emptyset$. Existence of finite, smaller elements guarantees existence of minimal elements: the set of all finite subsets of L is well-founded. In this case proposition 4.5 is still valid.

Proposition 4.7
If $S \in \mathcal{LTS}$ is image-finite, and $\overline{M_\epsilon}(S) \neq \emptyset$, then minimal elements of $\overline{M_\epsilon}(S)$ exist.

□

If S is not image-finite, minimal elements of $\overline{M_\epsilon}(S)$ need not exist. Optimizations of $\overline{M_\epsilon}(S)$ can be found by removing elements A if there is $A' \in \overline{M_\epsilon}(S)$ such that $A' \subseteq A$, but it may be that this optimization process never ends, like in example 4.6. We introduce a *reduced must set*, defined by the preorder $\sqsubseteq$ over sets of sets of actions, to formalize this notion of optimization. M_1 is a reduced must set of M_2, $M_1 \sqsubseteq M_2$, if M_1 is obtained from M_2 by a few optimization steps, i.e. by removing some elements from M_2, but such that only those elements are removed for which there is a smaller element ($rcl_\subseteq(M_1) = rcl_\subseteq(M_2)$), and such that no actions are removed ($\bigcup M_1 = \bigcup M_2$). (Cf. (4.5).) If a must set is right-closed, i.e. if $rcl(M) = M$, it is denoted by $\overline{M}$, like $\overline{M_\epsilon}(S)$.

Definition 4.8
Let $M_1, M_2 \in \mathcal{P}(\mathcal{P}(L))$, then

$$M_1 \sqsubseteq M_2 \;\; =_{def} \;\; M_1 \subseteq M_2 \text{ and } rcl_\subseteq(M_1) = rcl_\subseteq(M_2) \text{ and } \bigcup M_1 = \bigcup M_2$$

□

Proposition 4.9
$\sqsubseteq$ is a partial order.

□

The generalization of proposition 4.5.2 to cope with both infinite L and infinite behaviour, is proposition 4.10. It includes in every step the possibility to terminate the test case by having $A = \emptyset$, and $min_\subseteq$ is replaced by $\sqsubseteq$. Note that proposition 4.10 does not define a function: there are different possibilities for $M \sqsubseteq \overline{M_\epsilon}(S)$, and each M results in another, complete test suite.

Proposition 4.10
Let $S \in \mathcal{LTS}$, then $\Pi \in \mathcal{P}(\mathcal{DLTS})$ is a complete test suite for S with respect to **conf**, if $\exists M \sqsubseteq \overline{M_\epsilon}(S)$:

$$\begin{aligned}\Pi \;=\; \{\; t_A \in \mathcal{DLTS} \mid\; & t_A = \Sigma\{a; t_a \mid a \in A\},\\ & (\; A \in M \;\text{ and }\; v(t_A) = \mathbf{fail} \;)\\ & \text{or }\; (\; A = out(S) \;\text{ and }\; v(t_A) = \mathbf{pass} \;)\\ & \text{or }\; (\; A = \emptyset \;\text{ and }\; v(t_A) = \mathbf{pass} \;),\\ & \text{for each } a \in A : t_a \text{ is element of a complete test suite}\\ & \text{for } \mathbf{choice}\; S\; \mathbf{after}\; a \;\;\}\end{aligned}$$

□

A more algorithmic way to express this is easily derived:

Algorithm 4.11
Let $S \in \mathcal{LTS}$, then a complete test suite for S is obtained as follows:

1. compute $\overline{M_\epsilon}(S)$;
2. compute any $M \sqsubseteq \overline{M_\epsilon}(S)$;
3. determine all combinations of $A \subseteq L$ and verdict v, such that
$$\begin{aligned}& (\; A \in M \;\text{ and }\; v = \mathbf{fail} \;)\\ \text{or }\; & (\; A = out(S) \;\text{ and }\; v = \mathbf{pass} \;)\\ \text{or }\; & (\; A = \emptyset \;\text{ and }\; v = \mathbf{pass} \;);\end{aligned}$$
4. for each combination of A and v add the test case $t = \Sigma\{a; t_a \mid a \in A\}$ with $v(t) = v$ to the test suite;
5. for each t_a with $a \in A$ repeat the algorithm for the specification $\mathbf{choice}\; S\; \mathbf{after}\; a$.

□

If we wish to derive only a few sound test cases, it is sufficient to choose one or more sets $A \subset L$ with corresponding verdicts in step 3. Any choice provides a sound test case; the combination of all possible choices makes the test suite complete.

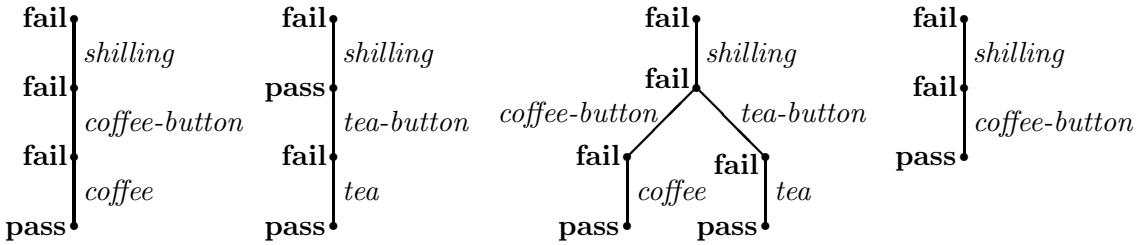


Figure 4.2: Test cases.

Example 4.12
Figure 4.2 gives some test cases for the vending machine M of figure 1.7 (section 1.4). The last test case is obtained using algorithm 4.11 as follows:

step 1,2: $M = \overline{M_\epsilon}(S) = \{\{shilling\}\}$;

step 3: $A = \{shilling\}$ and $v(t) = \mathbf{fail}$;

step 4,5: $t_{\{shilling\}} = shilling; t_{shilling}$,
with $t_{shilling}$ a sound test case for **choice** S **after** *shilling*
$= \mathbf{i}; (\ coffee\text{-}button; coffee; \mathbf{stop} \,\Box\, tea\text{-}button; tea; \mathbf{stop}\ )$
$\Box\ \mathbf{i}; coffee\text{-}button; coffee; \mathbf{stop}$;

step 1,2: $M = \overline{M_\epsilon}(\ \mathbf{choice}\ S\ \mathbf{after}\ shilling\ )$
$= \{\{coffee\text{-}button, tea\text{-}button\}, \{coffee\text{-}button\}\}$;

step 3: $A = \{coffee\text{-}button\}$ and $v(t) = \mathbf{fail}$;

step 4,5: $t_{shilling} = coffee\text{-}button; t_{coffee\text{-}button}$,
with $t_{coffee\text{-}button}$ a sound test case for
choice (**choice** S **after** *shilling*) **after** *coffee-button* $=$ $coffee; \mathbf{stop}$;

step 1,2: $M = \overline{M_\epsilon}(coffee; \mathbf{stop}) = \{\{coffee\}\}$;

step 3: $A = \emptyset$ and $v(t) = \mathbf{pass}$;

step 4,5: $t_{coffee\text{-}button} = \Sigma\emptyset = \mathbf{stop}$.

The second test case is obtained using proposition 4.5.1; it is not contained in any test suite derived according to proposition 4.5.2 or proposition 4.10.

□

4.2.1 The Canonical Tester

The test cases in the test suite of proposition 4.10 can also be expressed in $\mathcal{LTS}$, with passing a test case defined by $\mathbf{passes}_{\mathcal{N}}$ (definition 3.20): add a branch $\mathbf{i}; \mathbf{stop}$ in every state t **after** σ of the test case with verdict **pass**. Such an extra branch allows the test case to deadlock (t **after** σ **deadlocks**), thus also resulting in a successful test according to $\mathbf{passes}_{\mathcal{N}}$. Note that a terminal state in a test case in the test suite of proposition 4.10 always has verdict **pass**.

If test cases are written in $\mathcal{LTS}$ the nondeterminism can be used to combine different test cases. The choice between execution of different test cases is then expressed by a nondeterministic choice in one test case. This principle can be applied to combine all test cases into one. Using proposition 4.10 it is not difficult to derive such a test case. The resulting test case is the *canonical tester*, introduced in [Bri87]. In [Bri87] it is proved that the canonical tester of a specification is unique modulo $\approx_{te}$, and that it possesses a mirror property: the canonical tester of a canonical tester is equal modulo $\approx_{te}$ to the original specification (proposition 4.15).

Definition 4.13 ([Bri87])
Let $S \in \mathcal{LTS}$, then $C_S \in \mathcal{LTS}$ is a *canonical tester* of S, if

- $traces(C_S) = traces(S)$, and
- $\{C_S\}$ is complete for **conf** using $\mathbf{passes}_{\mathcal{N}}$.

□

Proposition 4.14
Let $Can(S) =_{def} \Sigma\{\ \mathbf{i}; \Sigma\{\ a; Can(\ \mathbf{choice}\ S\ \mathbf{after}\ a\ ) \mid a \in A\ \}\ \mid A \in M_S\ \}$,

with $\begin{cases} M_S \sqsubseteq \overline{M_\epsilon}(S) & \text{if } \overline{M_\epsilon}(S) \neq \emptyset \\ M_S = \{\ \emptyset, out(S)\ \} & \text{if } \overline{M_\epsilon}(S) = \emptyset \end{cases}$

then $Can(S)$ is a canonical tester of S.

Note that there may be many choices for M_S, hence $Can(S)$ is not uniquely determined.

□

Proposition 4.15 ([Bri87])

1. If $C_1, C_2 \in \mathcal{LTS}$ are both canonical testers of S, then $C_1 \approx_{te} C_2$.
2. If C_S is a canonical tester of S, and C_{C_S} is a canonical tester of C_S, then $C_{C_S} \approx_{te} S$.

□

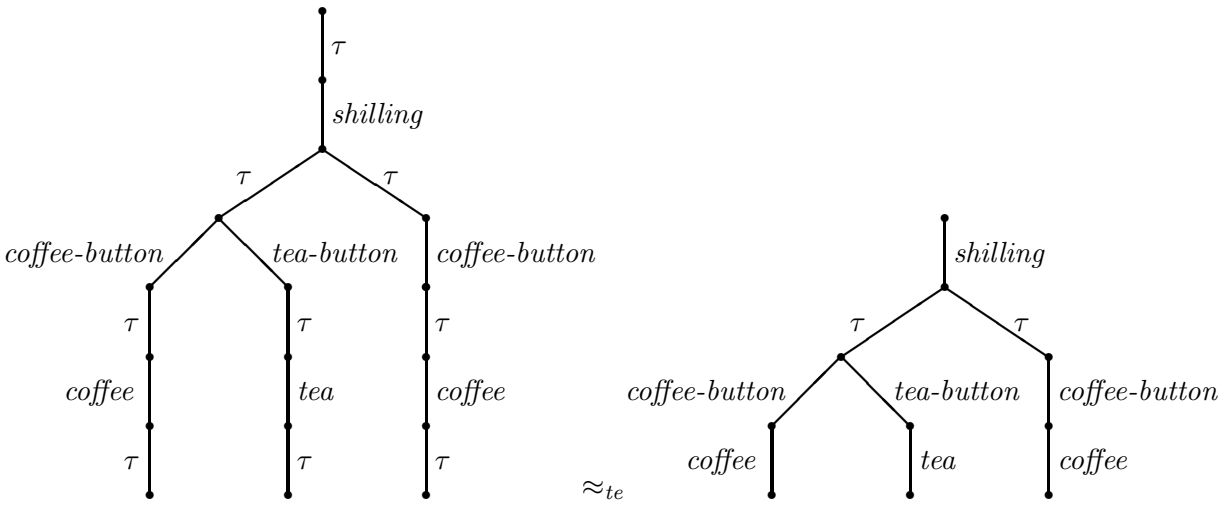


Figure 4.3: Canonical tester.

Example 4.16
Figure 4.3 gives the canonical tester of the vending machine of figure 1.7. The left process is obtained by following proposition 4.14 step by step; the right one is testing-equivalent $\approx_{te}$ to it.

□

4.3 Language Based Test Derivation

The theory of test derivation for labelled transition systems presented in section 4.2 is complete, in the sense that for each labelled transition system a test suite can be derived.

Since labelled transition systems form a semantical model for a number of behavioural specification formalisms, we now could derive a test suite from any expression in such a language by first deriving the labelled transition system forming its semantics, and then using one of the algorithms of section 4.2. However, such a labelled transition system is large, usually even infinite in the number of states and transitions. This means that it is practically unfeasible to use this labelled transition system for the derivation of tests, especially if we wish to implement the test derivation algorithms in executable programs. A finite representation of specification and test suite is then particularly needed. This problem can be solved if test suites are derived directly from a language expression, without explicitly deriving its labelled transition system semantics. Such an algorithm should be correct, i.e. the semantics of the resulting test case must be equivalent to a test case obtained from the semantics of the behaviour expression. This is expressed by the diagram in figure 4.4: it shall commute.

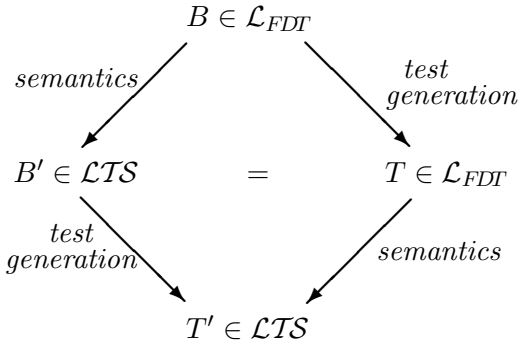


Figure 4.4: Test generation from behaviour expressions.

For easy test derivation from language expressions it is desirable that the algorithm is compositional: test cases, or at least attributes from which test cases can be derived, should be computed following the syntax of the expression. This section develops such a compositional test derivation algorithm. The language that is used is very simple, it is the language $\mathcal{BEX}$ defined in section 1.4 (definition 1.6), restricted to:

$$\mathbf{stop} \quad a;B \quad \mathbf{i};B \quad B \,\Box\, B \tag{4.6}$$

The algorithm is related to the CO-OP method [Wez90], which gives compositional rules for test derivation for the language Basic LOTOS [BB87]. In the next section our language is extended with the possibility to define an infinite number of transitions, i.e. infinite choice is added.

The starting point for language based test derivation is algorithm 4.11. It turns out that this algorithm cannot be applied directly to language expressions. We transform it by introducing *acceptance sets*, and we give compositional rules to compute these sets from language expressions.

4.3.1 Acceptance Sets

Analysis of algorithm 4.11 shows that the attributes needed to derive a test case are $M \sqsubseteq \overline{M_\epsilon}(S)$, $out(S)$, and **choice** S **after** a. While it turns out that $out(S)$ and **choice** S **after** a are easily determined from a behaviour expression, $M \sqsubseteq \overline{M_\epsilon}(S)$ poses more problems. It consists of sets that are defined using universal quantification over all states that can be reached after ϵ. This makes it difficult to determine these sets from subexpressions of a behaviour expression, e.g. for $\overline{M_\epsilon}(B)$ of $B = \mathbf{i}; a; \mathbf{stop} \,\Box\, \mathbf{i}; b; \mathbf{stop}$ information from both operands of $\Box$ must be used. The use of sets of actions that are refused, B **after** ϵ **refuses** A , solves this problem: these sets are defined using existential quantification over states. For convenience of representation we do not take these sets of refused actions, but their complement: the sets of actions that can occur in a state. We call them *acceptance sets*, and denote them by $\overline{C_\epsilon}$. Like with must sets, *reduced acceptance sets* are defined using the relation $\sqsubseteq$.

We show that on the one hand reduced acceptance sets are easily related to must sets using a transformation Ψ (proposition 4.18.3 and 4.18.4), so that they can be used as the basis for test derivation, while on the other hand they can be compositionally determined from behaviour expressions (proposition 4.24).

Definition 4.17

1. $\overline{C_\epsilon}(S) \;=_{def}\; \{\; out(S) \backslash A \mid S \textbf{ after } \epsilon \textbf{ refuses } A \;\}$
2. The function $\Psi : \mathcal{P}(\mathcal{P}(L)) \rightarrow \mathcal{P}(\mathcal{P}(L))$ is defined for $C \in \mathcal{P}(\mathcal{P}(L))$, by:
$$\Psi(C) \;=_{def}\; \{\; A \subseteq \bigcup C \mid \forall A' \in C : A \cap A' \neq \emptyset \;\}$$
3. $C_1 \in \mathcal{P}(\mathcal{P}(L))$ is a *reduced acceptance set* of $C_2 \in \mathcal{P}(\mathcal{P}(L))$, if $C_1 \sqsubseteq C_2$.

□

Proposition 4.18
Let $S \in \mathcal{LTS}$, then

1. $$\begin{array}{ll} & \{\; out(S') \mid S \stackrel{\epsilon}{\Longrightarrow} S' \stackrel{\tau}{\not\longrightarrow} \;\} \;\cup\; \{out(S)\} \\ \sqsubseteq & \{\; out(S') \mid S \stackrel{\epsilon}{\Longrightarrow} S' \;\} \\ \sqsubseteq & \{\; A \subseteq out(S) \mid \exists S' \;(\; S \stackrel{\epsilon}{\Longrightarrow} S' \text{ and } out(S') \subseteq A \;)\;\} \\ = & \overline{C_\epsilon}(S) \end{array}$$
2. If $C_1 \sqsubseteq C_2$ then $\Psi(C_1) = \Psi(C_2)$
3. Let $C \sqsubseteq \overline{C_\epsilon}(S)$, then $\Psi(C) = \overline{M_\epsilon}(S)$
4. Let $M \sqsubseteq \overline{M_\epsilon}(S)$, then $\Psi(M) = \overline{C_\epsilon}(S)$

□

The transformation $\Psi(C)$ defines a set of sets such that the elements have a non-empty intersection with a given set of sets C. An operational way to express this for finite C is 'form sets by taking one (or more) element from each set in C' (cf. the function *orth* in [PF90, Wez90]). This is formally expressed by the transformation Ψ_o (definition 4.19). Like for must sets, if C is finite a minimally reduced acceptance set can always be found:

finiteness is a sufficient condition for minimal elements to exist (cf. appendix A). The set $\Psi_o(C)$ may be infinite, but all elements are finite: $\Psi_o(C)$ is a subset of the set of all finite subsets of L. This implies that also for $\Psi_o(C)$ minimal elements exist (proposition A.2, cf. proposition 4.7). All this is applicable to behaviour expressions in $\mathcal{BEX}$ restricted to (4.6): such expressions are image-finite (definition 1.11.6), which is a sufficient condition for the existence of finite reduced acceptance sets. (Note that the full language $\mathcal{BEX}$ is not image-finite, e.g. $\Sigma\{\mathbf{i}; a_i; \mathbf{stop} \mid i \in \mathbf{N}\}$.)

Definition 4.19
Let $C = \{A_1, A_2, \ldots, A_n\}$ be a set or multi-set, then

$$\Psi_o(C) \;=_{def}\; \{\; \{a_1, a_2, \ldots, a_n\} \mid a_i \in A_i, 1 \leq i \leq n \;\}$$

□

Lemma 4.20

1. If $S \in \mathcal{LTS}$ is image-finite, then
   - there is a finite $C \sqsubseteq \overline{C_\epsilon}(S)$;
   - for any σ: **choice** S **after** σ is image-finite.
2. If $C \sqsubseteq \overline{C_\epsilon}(S)$ is finite, then
   - $\Psi_o(C) \sqsubseteq \Psi(C)$;
   - $min_\subseteq(C) \cup \{out(S)\} \;\sqsubseteq\; \overline{C_\epsilon}(S)$;
   - $min_\subseteq(\Psi_o(C)) \cup \{out(S) \mid \emptyset \notin C\} \;\sqsubseteq\; \Psi(\overline{C_\epsilon}(S))$.
3. The language $\mathcal{BEX}$ restricted to (4.6) is image-finite.

□

Now algorithm 4.11 is adapted in a straightforward way to the use of finite reduced acceptance sets. Proposition 4.18.3 and lemma 4.20 are used to compute a reduced must set.

Algorithm 4.21
Let $S \in \mathcal{LTS}$ be image-finite, then a sound test case $t \in \mathcal{DLTS}$ for S is obtained by:

1. choose $M \sqsubseteq \Psi_o(C)$, where $C \sqsubseteq \overline{C_\epsilon}(S)$;
2. choose: $A \in M$ and $v(t) = \mathbf{fail}$,
   or: if $M = \emptyset$ then $A = out(S)$ and $v(t) = \mathbf{pass}$
   or: $A = \emptyset$ and $v(t) = \mathbf{pass}$;
3. $t \;=\; \Sigma\{\; a; t_a \mid a \in A \;\}$, where t_a is a sound test case for **choice** S **after** a.

□

Remark 4.22
A reduced acceptance set is related to the set *Compulsory* of the CO-OP method in [Wez90]:

$$Compulsory(S) \;=_{def}\; \{\; out(S') \mid S \stackrel{\epsilon}{\Longrightarrow} S' \stackrel{\tau}{\not\longrightarrow} \;\}$$

the difference being that for a reduced acceptance set C always $\bigcup C = out(S)$, cf. the first characterization in proposition 4.18.1. Having this property the *Options* of the CO-OP method,

$$Options(S) \quad =_{def} \quad \{\ a \in L \mid \exists S'\ (\ S \stackrel{\epsilon}{\Longrightarrow} S' \stackrel{\tau}{\longrightarrow} \ \text{and}\ \ S' \stackrel{a}{\longrightarrow}\ )\ \}$$

are not needed anymore. It is another way of taking care that no actions 'are lost' (see the previous section).

As an example consider the process $S = a; \mathbf{stop} \,\Box\, \mathbf{i}; b; \mathbf{stop}$:

$$\begin{array}{rcl} Compulsory(S) & = & \{\{b\}\} \\ Options(S) & = & \{a\} \\ \overline{C_\epsilon}(S) & = & \{\{b\}, \{a, b\}\} \end{array}$$

$min_{\subseteq}(\overline{C_\epsilon}(S)) = Compulsory(S),\ \ out(S) \setminus \bigcup Compulsory(S) = Options(S).$ □

Remark 4.23
The subscript ϵ in $\overline{C_\epsilon}(S)$, and also in $\overline{M_\epsilon}(S)$, refers to the fact that we use acceptance and must sets after trace ϵ: $S \stackrel{\epsilon}{\Longrightarrow} S'$. In [Tre90] these sets were generalized for arbitrary σ, which gives a model for processes modulo $\approx_{te}$. □

4.3.2 Compositional Test Derivation

Now that we have acceptance sets as a way to derive test cases, compositional rules are defined with which acceptance sets can be compositionally derived from the restricted class of behaviour expressions in (4.6). We introduce $out_{\mathcal{B}}(B)$ and $\mathbf{choice}_{\mathcal{B}}\ B\ \mathbf{after}\ g$ as the counterparts of *out* and **choice** .. **after** .. for behaviour expressions. The rules for $out_{\mathcal{B}}(B)$ follow straightforwardly from the definition of *out*; for $\mathbf{choice}_{\mathcal{B}}\ B\ \mathbf{after}\ a$ proposition 4.3.1 is used as starting point, while a specific $C \sqsubseteq \overline{C_\epsilon}(B)$, denoted by $\mathcal{C}(B)$, is derived using the characterizations in proposition 4.18.1. Moreover, it turns out that in the compositional rule for $\Box$ information about the stability of the operands is required (definition 1.11.8). We denote stability of B by the predicate $st_{\mathcal{B}}(B)$, and add it as an extra attribute, which is to be computed compositionally.

Table 4.1 gives all rules, and proposition 4.24 states the correctness of the rules: for each attribute the value computed following table 4.1 coincides with applying the corresponding definition to the labelled transition system semantics. Using table 4.1 algorithm 4.21 can directly be applied. It will be evident that the canonical tester algorithm $Can(S)$ (proposition 4.14) can be adapted analogously.

Proposition 4.24
Let $B \in \mathcal{BEX}$, restricted to (4.6), and let $out_{\mathcal{B}}(B)$, $st_{\mathcal{B}}(B)$, $\mathcal{C}(B)$, and $\mathbf{choice}_{\mathcal{B}}\ B\ \mathbf{after}\ g$ be compositionally defined in table 4.1, then

- $out_{\mathcal{B}}(B) = out(\ell ts(B))$

- $st_{\mathcal{B}}(B)$ iff $\ell ts(B)$ is stable
- $\mathcal{C}(B) \sqsubseteq \overline{C_\epsilon}(\ell ts(B))$
- $\mathbf{choice}_{\mathcal{B}}\ B\ \mathbf{after}\ g \approx_{te} \mathbf{choice}\ \ell ts(B)\ \mathbf{after}\ g$

□

$\underline{B}$	$\underline{out_{\mathcal{B}}(B)}$	$\underline{st_{\mathcal{B}}(B)}$	$\underline{\mathcal{C}(B)}$	$\underline{\mathbf{choice}_{\mathcal{B}}\ B\ \mathbf{after}\ g}$
stop	$\emptyset$	*true*	$\{\emptyset\}$	**stop**
$a; B_1$	$\{a\}$	*true*	$\{\{a\}\}$	$\mathbf{i}; B_1$ if $a = g$ **stop** if $a \neq g$
$\mathbf{i}; B_1$	$out_{\mathcal{B}}(B_1)$	*false*	$\mathcal{C}(B_1)$	$\mathbf{choice}_{\mathcal{B}}\ B_1\ \mathbf{after}\ g$
$B_1 \,\square\, B_2$	$out_{\mathcal{B}}(B_1)$ $\cup out_{\mathcal{B}}(B_2)$	$st_{\mathcal{B}}(B_1)$ and $st_{\mathcal{B}}(B_2)$	$\{out_{\mathcal{B}}(B)\}$ $\cup$ if $\not st_{\mathcal{B}}(B_1)$ then $\mathcal{C}(B_1)$ $\cup$ if $\not st_{\mathcal{B}}(B_2)$ then $\mathcal{C}(B_2)$	$\mathbf{choice}_{\mathcal{B}}\ B_1\ \mathbf{after}\ g$ $\square\, \mathbf{choice}_{\mathcal{B}}\ B_2\ \mathbf{after}\ g$

Table 4.1: Compositional computation of acceptance sets.

Example 4.25
Consider the behaviour expression in example 1.8 representing the process of figure 1.7 (section 1.4):

$$\begin{aligned} B \;=\;& \mathit{shilling}; (\mathit{coffee\text{-}button}; \mathit{coffee}; \mathbf{stop} \,\square\, \mathit{tea\text{-}button}; \mathit{tea}; \mathbf{stop}) \\ & \square\, \mathit{shilling}; \mathit{coffee\text{-}button}; \mathit{coffee}; \mathbf{stop} \end{aligned}$$

The first test case in figure 4.2 is derived following algorithm 4.21 and proposition 4.24, as follows, using $B = B_1 \,\square\, B_2$, with:

$$\begin{aligned} B_1 &= \mathit{shilling}; (\mathit{coffee\text{-}button}; \mathit{coffee}; \mathbf{stop} \,\square\, \mathit{tea\text{-}button}; \mathit{tea}; \mathbf{stop}) \\ B_2 &= \mathit{shilling}; \mathit{coffee\text{-}button}; \mathit{coffee}; \mathbf{stop} \end{aligned}$$

step 1: $out_{\mathcal{B}}(B) = out_{\mathcal{B}}(B_1 \,\square\, B_2) = out_{\mathcal{B}}(B_1) \cup out_{\mathcal{B}}(B_2) = \{\mathit{shilling}\}$

$st_{\mathcal{B}}(B)$ iff $st_{\mathcal{B}}(B_1)$ and $st_{\mathcal{B}}(B_2)$ iff *true* and *true*

$\mathcal{C}(B) = \{out_{\mathcal{B}}(B)\} = \{\{\mathit{shilling}\}\}$

$\Psi_o(\mathcal{C}(B)) = \Psi_o(\{\{\mathit{shilling}\}\}) = \{\{\mathit{shilling}\}\}$

Choose $A = \{\mathit{shilling}\}$, and $v = \mathbf{fail}$.

$\mathbf{choice}_{\mathcal{B}}\ B\ \mathbf{after}\ \mathit{shilling}$
$= \mathbf{choice}_{\mathcal{B}}\ B_1\ \mathbf{after}\ \mathit{shilling} \;\square\; \mathbf{choice}_{\mathcal{B}}\ B_2\ \mathbf{after}\ \mathit{shilling}$
$= \mathbf{i}; (\mathit{coffee\text{-}button}; \mathit{coffee}; \mathbf{stop} \,\square\, \mathit{tea\text{-}button}; \mathit{tea}; \mathbf{stop})$
$\square\, \mathbf{i}; \mathit{coffee\text{-}button}; \mathit{coffee}; \mathbf{stop}$

step 2: Let $B' =$ **choice** B **after** *shilling* , then

$out_{\mathcal{B}}(B') = \{$*coffee-button*, *tea-button*$\}$, B' is not stable,

$$\begin{aligned}\mathcal{C}(B') &= \{out_{\mathcal{B}}(B')\} \\ &\quad \cup \mathcal{C}(\mathbf{i}; (\textit{coffee-button}; \textit{coffee}; \mathbf{stop} \,\Box\, \textit{tea-button}; \textit{tea}; \mathbf{stop})) \\ &\quad \cup \mathcal{C}(\mathbf{i}; \textit{coffee-button}; \textit{coffee}; \mathbf{stop}) \\ &= \{\{\textit{coffee-button}, \textit{tea-button}\}, \{\textit{coffee-button}\}\}\end{aligned}$$

$$\begin{aligned}\Psi_o(\mathcal{C}(B')) &= \Psi_o(\{\{\textit{coffee-button}, \textit{tea-button}\}, \{\textit{coffee-button}\}\}) \\ &= \{\{\textit{coffee-button}, \textit{tea-button}\}, \{\textit{coffee-button}\}\}\end{aligned}$$

Choose $A = \{$*coffee-button*$\}$ and $v =$ **fail**.

$$\begin{aligned}&\mathbf{choice}_{\mathcal{B}}\ B' \ \mathbf{after}\ \textit{coffee-button} \\ =\ & \mathbf{choice}_{\mathcal{B}}\ \mathbf{i}; (\textit{coffee-button}; \textit{coffee}; \mathbf{stop} \Box \textit{tea-button}; \textit{tea}; \mathbf{stop}) \\ & \quad \mathbf{after}\ \textit{coffee-button} \\ & \Box\ \mathbf{choice}_{\mathcal{B}}\ \mathbf{i}; \textit{coffee-button}; \textit{coffee}; \mathbf{stop}\ \mathbf{after}\ \textit{coffee-button} \\ =\ & \mathbf{choice}_{\mathcal{B}}\ (\textit{coffee-button}; \textit{coffee}; \mathbf{stop} \Box \textit{tea-button}; \textit{tea}; \mathbf{stop}) \\ & \quad \mathbf{after}\ \textit{coffee-button} \\ & \Box\ \mathbf{choice}_{\mathcal{B}}\ \textit{coffee-button}; \textit{coffee}; \mathbf{stop}\ \mathbf{after}\ \textit{coffee-button} \\ =\ & \mathbf{choice}_{\mathcal{B}}\ \textit{coffee-button}; \textit{coffee}; \mathbf{stop}\ \mathbf{after}\ \textit{coffee-button} \\ & \Box\ \mathbf{choice}_{\mathcal{B}}\ \textit{tea-button}; \textit{tea}; \mathbf{stop}\ \mathbf{after}\ \textit{coffee-button} \\ & \Box\ \mathbf{i}; \textit{coffee}; \mathbf{stop} \\ =\ & \mathbf{i}; \textit{coffee}; \mathbf{stop} \Box \mathbf{stop} \Box \mathbf{i}; \textit{coffee}; \mathbf{stop} \\ =\ & \mathbf{i}; \textit{coffee}; \mathbf{stop}\end{aligned}$$

step 3: Let $B'' =$ $\mathbf{choice}_{\mathcal{B}}\ B'$ **after** *coffee-button* $= \mathbf{i}; \textit{coffee}; \mathbf{stop}$, then

$out_{\mathcal{B}}(B'') = \{$*coffee*$\}$, B'' is not stable, $\mathcal{C}(B'') = \{\{\textit{coffee}\}\}$, $\Psi_o(\mathcal{C}(B'')) = \{\{\textit{coffee}\}\}$, and $\mathbf{choice}_{\mathcal{B}}\ B''$ **after** *coffee* $= \mathbf{i}; \mathbf{stop}$.

Choose $A = \{$*coffee*$\}$, and $v =$ **fail**.

step 4: Let $B''' =$ $\mathbf{choice}_{\mathcal{B}}\ B''$ **after** *coffee* $= \mathbf{i}; \mathbf{stop}$, then
$out_{\mathcal{B}}(B''') = \emptyset$, $\mathcal{C}(B''') = \{\emptyset\}$, and $\Psi_o(\mathcal{C}(B''')) = \emptyset$.

$A = out_{\mathcal{B}}(B''') = \emptyset$, and $v =$ **pass**.

□

Compositional rules for the other syntactic constructs of $\mathcal{BEX}$ can be derived analogously. As far as $\mathcal{BEX}$ coincides with Basic LOTOS the related attributes *Compulsory* and *Options* (remark 4.22) can be found in [Wez90].

The rules in table 4.1 can be interpreted as attribute rules defining values for the synthesized attributes $out(B)$, etc. in an attribute grammar based on $\mathcal{BEX}$. Using the rules of the CO-OP method such an attribute grammar was developed in [Ald90], on the basis of which COOPER was generated, a tool that derives tests from Basic LOTOS expressions.

4.4 Test Derivation with Infinite Branching

In the previous section we studied test derivation from behaviour expressions, where, in principle, we could have first derived the labelled transition system semantics, and then applied a test derivation algorithm for labelled transition systems. We now extend our language such that infinitely branching labelled transition systems can be defined. First this language is defined, then we consider the initial behaviour of test cases, i.e. how to determine the set A in the test case $\Sigma\{a;t_a \mid a \in A\}$ of the test derivation algorithms (e.g. algorithm 4.21), and finally the subsequent behaviour of a tester, i.e. the part t_a, is studied.

A Language with Infinite Branching

We define a language $\mathcal{BEX}_v$ that allows infinite branching. This is accomplished by introducing actions consisting of a pair of a *gate* and a *value*. Let G be the set of gates, which we assume to be finite, and let V be the set of possible values, not necessarily finite, then the set of observable actions L is $G \times V$. Infinite branching is introduced by having variables over these values in behaviour expressions. Let x be a variable from a domain of variables X, and let g be a gate, then

$$g?x;B$$

represents a labelled transition system that can make a transition $\langle g, v\rangle$ to B for any value v that x can have. The expression is equivalent to

$$\Sigma\ \{\ \langle g, v\rangle; B \mid v \in V\ \}$$

The values that x may have can be restricted by a *predicate*. Let p be a predicate in which x occurs as a free variable, e.g. $x > 5$, then

$$g?x : x > 5; B \quad = \quad \Sigma\ \{\ \langle g, v\rangle; B \mid v \in V, v > 5\ \}$$

The variable x introduced in the expression $g?x;B$, may occur as a free variable in the subsequent behaviour B, i.e. B may depend on the value of x. The value of variable x in B is determined by the transition that $g?x;B$ makes in a particular execution:

$$g?x;B \ = \ \Sigma\ \{\ \langle g, v\rangle; B \mid v \in V\ \} \xrightarrow{\langle g,v\rangle} B[v/x]$$

$B[v/x]$ denotes the expression B where each free occurrence of x is replaced by the value v.

The usual interpretation of actions consisting of a gate and a value is that of *value communication*. A gate represents a *place* where communication occurs, e.g. a SAP (Service Access Point) of a protocol, or a PCO (see sections 1.3.3 and 2.2.6). A value represents the message that is communicated at that place. This is the kind of communication as it occurs for instance in the formal description technique LOTOS.

For the language $\mathcal{BEX}_v$, offering this kind of infinite branching, we restrict the values to natural numbers, denoted by $\mathbf{N}$, and we assume the existence of a class of predicates P over the natural numbers. We refrain from a precise formal definition of P, but we assume that all predicates are effectively computable. They may contain standard relations like $=, <, \leq, \ldots$, constants $0, 1, 2, \ldots$, and arithmetic operators $+, -, \ldots$. Moreover, variables $x, y, \ldots$ in X can be used in predicates. The variables used in a predicate $p \in P$ are the *free variables* of that predicate, denoted by $FV(p)$. A *closed predicate* has no free variables: $FV(p) = \emptyset$. The *substitution* of a value v for a variable x in a predicate p is denoted by $p[v/x]$. It is the result of the replacement of all free occurrences of x in p by v. The semantics of a closed predicate is a boolean value, either *true* or *false*, obtained using the usual evaluation rules for natural numbers.

Definition 4.26
Let G be a finite set of *gates*, X a set of *variables*, P a set of *predicate expressions* over $\mathbf{N}$, with $FV(p)$ denoting the free variables of a predicate $p \in P$, and $p[v/x]$ denoting the substitution of a variable $x \in X$ in p by a value $v \in \mathbf{N}$. Let g range over G, p over P, x over X, and v over $\mathbf{N}$.

1. The syntax of the language $\mathcal{BEX}_v$ is defined by

$$B \quad =_{def} \quad \mathbf{stop} \quad | \quad g?x : p; B \quad | \quad \mathbf{i} : p; B \quad | \quad B \,\Box\, B$$

2. The *free variables* of $B \in \mathcal{BEX}_v$, $FV(B)$, are:

$$\begin{array}{lll} FV(\mathbf{stop}) & =_{def} & \emptyset \\ FV(g?x : p; B) & =_{def} & (FV(B) \cup FV(p)\ ) \backslash \{x\} \\ FV(\mathbf{i} : p; B) & =_{def} & FV(B) \cup FV(p) \\ FV(B_1 \,\Box\, B_2) & =_{def} & FV(B_1) \cup FV(B_2) \end{array}$$

3. The substitution $B[v/y]$ is the behaviour expression defined by:

$$\begin{array}{lll} \mathbf{stop}[v/y] & =_{def} & \mathbf{stop} \\ (g?x : p; B)[v/y] & =_{def} & \begin{cases} g?x : p; B & \text{if } x = y \\ g?x : p[v/y]; B[v/y] & \text{if } x \neq y \end{cases} \\ (\mathbf{i} : p; B)[v/y] & =_{def} & \mathbf{i} : p[v/y]; B[v/y] \\ (B_1 \,\Box\, B_2)[v/y] & =_{def} & B_1[v/y] \,\Box\, B_2[v/y] \end{array}$$

4. A behaviour expression $B \in \mathcal{BEX}_v$ is *closed*, if $FV(B) = \emptyset$; otherwise B is an *open* behaviour expression. The set of closed behaviour expressions is denoted by $\mathcal{BEX}_v^c$.

5. The semantics of a closed behaviour expression $B \in \mathcal{BEX}_v^c$ is defined using an interpretation $\Im$ of $\mathcal{BEX}_v^c$ in $\mathcal{BEX}$. The semantics of B is the labelled transition system $\ell ts(\Im(B))$, where

$$\begin{array}{lll} \Im(\mathbf{stop}) & =_{def} & \mathbf{stop} \\ \Im(g?x : p; B) & =_{def} & \Sigma\{\langle g, v\rangle; \Im(B[v/x]) \mid p[v/x]\} \\ \Im(\mathbf{i} : p; B) & =_{def} & \begin{cases} \mathbf{i}; \Im(B) & \text{if } p \\ \mathbf{stop} & \text{if not } p \end{cases} \\ \Im(B_1 \Box B_2) & =_{def} & \Im(B_1) \Box \Im(B_2) \end{array}$$

6. We introduce the following notation:
   - Let E be an expression denoting a natural number, and let y be a variable occurring nowhere in the behaviour expression under consideration, then $g!E; B \ =_{def} \ g?y : y = E; B$.
   - $g?x; B \ =_{def} \ g?x : true; B$
   - Like for $\mathcal{BEX}$ ';' binds stronger than '$\Box$'.

□

Example 4.27
An example of a behaviour expression in $\mathcal{BEX}_v^c$ is

$$\begin{array}{rll} B = & & g?x : x < 20; h!x + 2; \mathbf{stop} \\ & \Box & \mathbf{i}; g?y : y > 10; h!y; \mathbf{stop} \end{array}$$

□

Initial Behaviour of a Test Case

This subsection deals with the first step in test case derivation: the initial behaviour of a test case. It involves determining a set $A \subseteq L$ such that $A \in M \sqsubseteq \overline{M_\epsilon}(S)$, or expressed in acceptance sets: $A \in M \sqsubseteq \Psi(C)$ where $C \sqsubseteq \overline{C_\epsilon}(S)$. We consider closed behaviour expressions only. Note that closedness of $g?x : p; B$ implies that x is the only free variable that can occur in p and B: $p[v/x]$ and $B[v/x]$ are closed.

Test derivation from expressions in $\mathcal{BEX}_v$ suffers from the problem of infiniteness. Because of the infinite branching, elements of acceptance sets turn out to be infinite, hence difficult to represent in automated algorithms. A first help in deriving test cases is proposition 4.28, which implies that finite reduced acceptance sets exist (lemma 4.20 and proposition 4.7).

Proposition 4.28
Any $B \in \mathcal{BEX}_v^c$ is image-finite.

□

Although finite reduced acceptance sets exist, the elements of acceptance sets, also sets, need not be finite. The next example introduces the *acceptance division lemma* (lemma 4.30) that provides a handle on how to deal with these infinite sets.

Example 4.29
Let $B \ = \ \mathbf{i}; (f?x; \mathbf{stop} \ \Box \ g?x; \mathbf{stop}) \ \Box \ \mathbf{i}; (g?x; \mathbf{stop} \ \Box \ h?x; \mathbf{stop})$

A reduced acceptance set $C(B)$ according to proposition 4.18 is

$$\begin{array}{rcl} C(B) & = & \{ \ out(B') \mid B \stackrel{\epsilon}{\Longrightarrow} B' \ \} \\ & = & \{ \ out(B), out(f?x; \mathbf{stop} \ \Box \ g?x; \mathbf{stop}), out(g?x; \mathbf{stop} \ \Box \ h?x; \mathbf{stop}) \ \} \end{array}$$

$$\begin{array}{rcl} \text{Let } F & = & \{\langle f, v\rangle \mid v \in \mathbf{N}\}, \\ G & = & \{\langle g, w\rangle \mid w \in \mathbf{N}\}, \\ H & = & \{\langle h, u\rangle \mid u \in \mathbf{N}\}, \end{array}$$

then $C(B)$ can be written as

$$C(B) = \{\ F \cup G \cup H, F \cup G,\ G \cup H\ \}$$

A reduced must set can be obtained using Ψ_o (definition 4.19 and lemma 4.20):

$$\begin{array}{rcl}
\Psi_o(C(B)) & = & \{\ \{a,b,c\} \mid a \in\ F \cup G \cup H,\ b \in F \cup G,\ c \in G \cup H\ \} \\
& = & \quad \{\ \{a,b,c\} \mid a \in F,\ b \in F,\ c \in G\ \} \\
& & \cup \{\ \{a,b,c\} \mid a \in F,\ b \in F,\ c \in H\ \} \\
& & \cup \{\ \{a,b,c\} \mid a \in F,\ b \in G,\ c \in G\ \} \\
& & \cup \{\ \{a,b,c\} \mid a \in F,\ b \in G,\ c \in H\ \} \\
& & \cup \{\ \{a,b,c\} \mid a \in G,\ b \in F,\ c \in G\ \} \\
& & \cup \{\ \{a,b,c\} \mid a \in G,\ b \in F,\ c \in H\ \} \\
& & \cup \{\ \{a,b,c\} \mid a \in G,\ b \in G,\ c \in G\ \} \\
& & \cup \{\ \{a,b,c\} \mid a \in G,\ b \in G,\ c \in H\ \} \\
& & \cup \{\ \{a,b,c\} \mid a \in H,\ b \in F,\ c \in G\ \} \\
& & \cup \{\ \{a,b,c\} \mid a \in H,\ b \in F,\ c \in H\ \} \\
& & \cup \{\ \{a,b,c\} \mid a \in H,\ b \in G,\ c \in G\ \} \\
& & \cup \{\ \{a,b,c\} \mid a \in H,\ b \in G,\ c \in H\ \}
\end{array}$$

This rather complex expression can be further reduced, using the properties of $\sqsubseteq$, but such reductions can be performed only on an ad hoc basis. We do not consider such reductions now, but we want to simplify the expression using the following observation:

$$\{\ \{a,b,c\} \mid a \in\ F,\ b \in F,\ c \in G\ \} \ =\ \Psi_o(\{F,F,G\})$$

and analogously for the other sets. Note that $\{F,F,G\}$ is treated as a multiset. Now $\Psi_o(C(B))$ can be rewritten as follows:

$$\begin{array}{rcl}
\Psi_o(C(B)) & = & \quad \Psi_o(\{F,F,G\}) \\
& & \cup\ \Psi_o(\{F,F,H\}) \\
& & \cup\ \Psi_o(\{F,G,G\}) \\
& & \cup\ \Psi_o(\{F,G,H\}) \\
& & \cup\ \Psi_o(\{G,F,G\}) \\
& & \cup\ \Psi_o(\{G,F,H\}) \\
& & \cup\ \Psi_o(\{G,G,G\}) \\
& & \cup\ \Psi_o(\{G,G,H\}) \\
& & \cup\ \Psi_o(\{H,F,G\}) \\
& & \cup\ \Psi_o(\{H,F,H\}) \\
& & \cup\ \Psi_o(\{H,G,G\}) \\
& & \cup\ \Psi_o(\{H,G,H\}) \\
& = & \bigcup\{\ \Psi_o(E) \mid E \in\ \Psi_o(\{\{F,G,H\},\{F,G\},\{G,H\}\})
\end{array}$$

We see that for the computation of a reduced acceptance set C of B we can consider each of the sets F, G, and H as a whole, compute Ψ_o as if these sets were actions, and then applying the transformation Ψ_o again to the results of the first step, which are now

sets of sets of actions. Since the sets F, G, and H exactly correspond to the actions induced by $f?x$, $g?x$, and $h?x$ respectively, this in fact means that Ψ_o can be computed symbolically on $f?x$, $g?x$, and $h?x$. This is formally expressed in the *acceptance division lemma* (lemma 4.30).

□

Lemma 4.30 (Acceptance Division Lemma)
Let $C \in \mathcal{P}(\mathcal{P}(L))$, $D \in \mathcal{P}(\mathcal{P}(\mathcal{P}(L)))$, such that there is a bijection $\delta : C \to D$, with for all $A \in C : \bigcup \delta(A) = A$, then

$$\Psi(C) \quad = \quad \bigcup \{\ \Psi(E) \mid E \in \Psi(D)\ \}$$

□

The acceptance division lemma expresses that for the computation of $\Psi(C)$, elements of the acceptance set C can be divided into subsets: let $A_1 \in C$, then A_1 is represented by the union of the elements $D'' \in D'_1$. The transformation Ψ is applied to C with these subsets D'' treated as elements. In the next step Ψ is applied to the sets of elements formed in the first step (E in the lemma). This means that if we can find a finite division of each element of an acceptance set, we can compute $\Psi(C)$ in two steps.

$$C = \{\underbrace{\{a_{1_1}, a_{1_2}, a_{1_3}, \ldots}_{A_1 = \bigcup_{\delta(A_1)=D'_1}} \}, \underbrace{\{a_{2_1}, a_{2_2}, a_{2_3}, \ldots}_{A_2 = \bigcup_{\delta(A_2)=D'_2}} \}, \ldots\}$$

$$D = \{\underbrace{\{\{d_{1_{1_1}}, d_{1_{1_2}}, \ldots\}, \{d_{1_{2_1}}, d_{1_{2_2}}, \ldots\}, \ldots\}}_{D''_1}, \underbrace{\{\{d_{2_{1_1}}, d_{2_{1_2}}, \ldots\}, \{d_{2_{2_1}}, d_{2_{2_2}}, \ldots\}, \ldots\}}_{D''_2}, \ldots\}$$

Reduced versions of the lemma can be derived, where not $\Psi(C)$ is computed but a reduced must set $M \sqsubseteq \Psi(C)$.

Lemma 4.30 is useful for $\mathcal{BEX}_v$, since it can facilitate computation of acceptance sets. Acceptance sets are in a natural way structured as required by lemma 4.30: they consist of sets of sets of actions: each element is formed by a finite number of subsets, each subset being a set $\{\langle g, v\rangle \mid p[v/x]\}$, corresponding to $g?x : p$ of the syntax. The lemma expresses that in the computation of $\Psi(C)$, the sets $\{\langle g, v\rangle \mid p[v/x]\}$ can be treated as a whole. Denoting this set $\{\langle g, v\rangle \mid p[v/x]\}$ by $g?x : p$, the relation to the previous picture is as follows:

$$D = \{\underbrace{\{\{\langle g_1, v_1\rangle, \langle g_1, v_2\rangle, \ldots\}, \{\langle g_2, w_1\rangle, \langle g_2, w_2\rangle, \ldots\}, \ldots\}}_{g_1?x:p}, \underbrace{\{\{\langle g_3, u_1\rangle, \ldots\}, \ldots\}}_{g_2?y:q}, \ldots\}$$

Computing $\Psi(C)$ by applying lemma 4.30 first involves computation of $\Psi(D)$. The result has the same structure as D, i.e. a set of sets of $g?x : p$, which is a set of sets of sets of actions. The next step involves for each $E \in \Psi(D)$ computation of $\Psi(E)$. This set E has a simpler structure: it is a set of $g?x : p$, i.e. a set of sets of actions. And since this set is always finite, Ψ_o can replace Ψ:

Let $E = \{\ g_1?x_1 : p_1,\ g_2?x_2 : p_2,\ \ldots,\ g_n?x_n : p_n\ \}$, then

$$\begin{array}{ll} & \Psi(E) \\ \sqsupseteq & \Psi_o(E) \\ = & \{\ \{\ \langle g_1, v_1\rangle,\ \langle g_2, v_2\rangle,\ \ldots,\ \langle g_n, v_n\rangle\ \} \mid p_1[v_1/x_1],\ p_2[v_2/x_2],\ \ldots,\ p_n[v_n/x_n]\ \} \end{array}$$

For the derivation of test cases an $A \in \Psi_o(E)$ is required. Such an A is always finite, and has the form $\{\ \langle g_1, v_1\rangle,\ \langle g_2, v_2\rangle,\ \ldots,\ \langle g_n, v_n\rangle\ \}$. The corresponding test case is

$$\Sigma\{a; t_a \mid a \in \{\ \langle g_1, v_1\rangle,\ \langle g_2, v_2\rangle,\ \ldots,\ \langle g_n, v_n\rangle\ \}\ \}$$

which can be written as

$$g_1!v_1; t_{\langle g_1, v_1\rangle}\ \Box\ g_2!v_2; t_{\langle g_2, v_2\rangle}\ \Box\ \ldots\ \Box\ g_n!v_n; t_{\langle g_n, v_n\rangle} \tag{4.7}$$

Also the canonical tester can be computed using lemma 4.30 (definition 4.13 and proposition 4.14). In this case a nondeterministic choice over all $A \in M = \Psi_o(C)$ has to be constructed. Since M itself may be infinite, this implies that a representation must be defined for such an infinite nondeterministic choice. In fact the notation † defined in [Doo91] is such a representation. Note that infinite nondeterministic choice implies that the canonical tester is not image finite. Hence the canonical tester of a behaviour expression in $\mathcal{BEX}_v$ cannot be expressed in $\mathcal{BEX}_v$.

Finally, compositional rules for the attributes $out_{\mathcal{D}}(B)$, $st_{\mathcal{D}}(B)$, and $\mathcal{D}(B)$ are given in table 4.2 (without proof). $out_{\mathcal{D}}(B)$ is a set of $g?x : p$, $st_{\mathcal{D}}(B)$ depends on the value of predicates in the behaviour expression, and $\mathcal{D}(B)$ gives a set of sets of $g?x : p$ as required by lemma 4.30 and discussed above.

Subsequent Behaviour of a Test Case

From equation 4.7 it follows that for the subsequent behaviour of test cases it is sufficient to consider a finite number of test cases t_a, where $t_a =$ **choice** B **after** $\langle g, v\rangle$. Table 4.3 gives compositional rules for **choice**$_{\mathcal{D}}$ B **after** $\langle g, v\rangle$, which is analogous to **choice**$_{\mathcal{B}}$ B **after** g in proposition 4.24.

Note that **choice**$_{\mathcal{D}}$ B **after** $\langle g, v\rangle$ is not easily applicable to the derivation of the canonical tester. Due to the infinity of M, **choice**$_{\mathcal{D}}$ B **after** $\langle g, v\rangle$ has to be determined for infinitely many $\langle g, v\rangle$. An expression of the form **choice** B **after** $g?x : p$ is needed to cope with this problem.

$\underline{B}$	$\underline{out_{\mathcal{D}}(B)}$	$\underline{st_{\mathcal{D}}(B)}$	$\underline{\mathcal{D}(B)}$
stop	$\emptyset$	*true*	$\{\emptyset\}$
$g?x : p; B_1$	$\{g?x : p\}$	*true*	$\{\{g?x : p\}\}$
$\mathbf{i} : p; B_1$	$out_{\mathcal{D}}(B_1)$ if p $\emptyset$ if not p	not p	$\mathcal{D}(B_1)$ if p $\{\emptyset\}$ if not p
$B_1 \,\square\, B_2$	$out_{\mathcal{D}}(B_1)$ $\cup out_{\mathcal{D}}(B_2)$	$st_{\mathcal{D}}(B_1)$ and $st_{\mathcal{D}}(B_2)$	$\{out_{\mathcal{D}}(B)\}$ $\cup$ if $\not st_{\mathcal{D}}(B_1)$ then $\mathcal{D}(B_1)$ $\cup$ if $\not st_{\mathcal{D}}(B_2)$ then $\mathcal{D}(B_2)$

Table 4.2: Compositional computation of acceptance sets for $\mathcal{BEX}_v$.

$\underline{B}$	$\underline{\mathbf{choice}_{\mathcal{D}}\ B\ \mathbf{after}\ \langle h, v\rangle}$
stop	**stop**
$g?x : p; B_1$	$\mathbf{i}; B_1[v/x]$ if $g = h$ and $p[v/x]$ **stop** if $g \neq h$ or not $p[v/x]$
$\mathbf{i} : p; B_1$	$\mathbf{choice}_{\mathcal{D}}\ B_1\ \mathbf{after}\ \langle h, v\rangle$ if p **stop** if not p
$B_1 \,\square\, B_2$	$\mathbf{choice}_{\mathcal{D}}\ B_1\ \mathbf{after}\ \langle h, v\rangle$ $\square\ \mathbf{choice}_{\mathcal{D}}\ B_2\ \mathbf{after}\ \langle h, v\rangle$

Table 4.3: Compositional computation of subsequent behaviour for $\mathcal{BEX}_v$.

Example 4.31
A test case for B in example 4.27 is derived in detail:

$$\begin{array}{rl} B = & g?x : x < 20; h!x + 2; \mathbf{stop} \\ \Box & \mathbf{i}; g?y : y > 10; h!y; \mathbf{stop} \end{array}$$

The first step:

$$\begin{array}{ll} & out_{\mathcal{D}}(B) \\ = & out_{\mathcal{D}}(g?x : x < 20; h!x + 2; \mathbf{stop}) \cup out_{\mathcal{D}}(\mathbf{i}; g?y : y > 10; h!y; \mathbf{stop}) \\ = & \{g?x : x < 20, g?y : y > 10\} \end{array}$$

$st_{\mathcal{D}}(g?x : x < 20; h!x + 2; \mathbf{stop}) = \mathit{true}$
$st_{\mathcal{D}}(\mathbf{i}; g?y : y > 10; h!y; \mathbf{stop}) = st_{\mathcal{D}}(\mathbf{i} : \mathit{true}; g?y : y > 10; h!y; \mathbf{stop}) = \mathit{false}$

$$\begin{array}{ll} & \mathcal{D}(B) \\ = & \{out_{\mathcal{D}}(B)\} \cup \mathcal{D}(\mathbf{i}; g?y : y > 10; h!y; \mathbf{stop}) \\ = & \{out_{\mathcal{D}}(B)\} \cup \mathcal{D}(g?y : y > 10; h!y; \mathbf{stop}) \\ = & \{\{g?x : x < 20, g?y : y > 10\}\} \cup \{\{g?y : y > 10\}\} \\ = & \{\{g?x : x < 20, g?y : y > 10\}, \{g?y : y > 10\}\} \end{array}$$

For one test case choose $E \in \Psi(\mathcal{D}(B))$ and $A \in \Psi_o(E)$:
$\Psi(\mathcal{D}(B)) = \{\{g?x : x < 20, g?y : y > 10\}, \{g?y : y > 10\}\}$, take $E = \{g?y : y > 10\}$,
and $A \in \Psi_o(E) : A = \{\langle g, 15\rangle\}$.

The test case then is: $t = g!15; t_{\langle g,15\rangle}$,
where $t_{\langle g,15\rangle}$ is a test case for **choice** B **after** $\langle g, 15\rangle$,
computed via **choice**$_{\mathcal{D}}$ B **after** $\langle g, 15\rangle$ in table 4.3:

$$\begin{array}{ll} & \mathbf{choice}_{\mathcal{D}}\ B\ \mathbf{after}\ \langle g, 15\rangle \\ = & \quad \mathbf{choice}_{\mathcal{D}}\ g?x : x < 20; h!x + 2; \mathbf{stop}\ \mathbf{after}\ \langle g, 15\rangle \\ & \Box\ \mathbf{choice}_{\mathcal{D}}\ \mathbf{i}; g?y : y > 10; h!y; \mathbf{stop}\ \mathbf{after}\ \langle g, 15\rangle \\ = & \mathbf{i}; h!15 + 2; \mathbf{stop}\ \Box\ \mathbf{choice}_{\mathcal{D}}\ g?y : y > 10; h!y; \mathbf{stop}\ \mathbf{after}\ \langle g, 15\rangle \\ = & \mathbf{i}; h!17; \mathbf{stop}\ \Box\ \mathbf{i}; h!15; \mathbf{stop} \end{array}$$

Now repeat the whole test derivation procedure for

$$B' = \mathbf{i}; h!17; \mathbf{stop}\ \Box\ \mathbf{i}; h!15; \mathbf{stop}$$

$out_{\mathcal{D}}(B') = \{h!17, h!15\}$
$st_{\mathcal{B}}(\mathbf{i}; h!17; \mathbf{stop}) = st_{\mathcal{B}}(\mathbf{i}; h!15; \mathbf{stop}) = \mathit{false}$

$$\begin{array}{ll} & \mathcal{D}(B) \\ = & out_{\mathcal{D}}(B') \cup out_{\mathcal{B}}(\mathbf{i}; h!17; \mathbf{stop}) \cup out_{\mathcal{B}}(\mathbf{i}; h!15; \mathbf{stop}) \\ = & \{\{h!17, h!15\}, \{h!17\}, \{h!15\}\} \end{array}$$

Take $E \in \Psi(\mathcal{D}(B))$, where $\Psi(\mathcal{D}(B)) = \{\{h!17, h!15\}\} : E = \{h!17, h!15\}$,
and $A \in \Psi_o(E) : A = \{\langle h, 15\rangle, \langle h, 17\rangle\}$.

The test case $t_{\langle g,15\rangle}$ is:

$$\begin{aligned} & \Sigma\{a; t_a \mid a \in \{\langle h, 15\rangle, \langle h, 17\rangle\}\} \\ = \; & \langle h, 15\rangle; t_{\langle h,15\rangle} \;\Box\; \langle h, 17\rangle; t_{\langle h,17\rangle} \\ = \; & h!15; t_{\langle h,15\rangle} \;\Box\; h!17; t_{\langle h,17\rangle} \end{aligned}$$

where $t_{\langle h,15\rangle}$ is a test case for **choice** B' **after** $\langle h, 15\rangle$, and $t_{\langle h,17\rangle}$ is a test case for **choice** B' **after** $\langle h, 17\rangle$. It is easy to check that both expressions are equal to $\mathbf{i}; \mathbf{stop}$, with test case **stop**.

Taking the parts together a test case for B is:

$$t \;=\; g!15;\; (\; h!15; \mathbf{stop} \;\Box\; h!17; \mathbf{stop} \;)$$

□

4.5 Concluding Remarks

We have worked on some of the problems of test derivation for the implementation relation **conf**, first from labelled transition systems, then from two simple languages with labelled transition system semantics. A main issue was representation of infinite choice in a finite manner to obtain implementable algorithms.

The algorithms for the derivation of test cases do not define functions, in the sense that choices within the algorithms result in different test cases. No choice is prescribed by the algorithms; they only guarantee soundness of the derived test case for each choice. Making a particular choice is a problem of *test case selection*; it could be called *horizontal test selection*. Test selection is studied in chapter 6.

Although the languages for behaviour expressions used in this chapter are rather simple they have many features also appearing in more complex languages with labelled transition system semantics, e.g. LOTOS [BB87, ISO89b]. Aspects not covered by $\mathcal{BEX}$ or $\mathcal{BEX}_v$ are recursion and image-infiniteness.

It was suggested to approach recursion, i.e. infinite behaviour, using the approximation induction principle: infinite behaviour is approximated by finite behaviour of arbitrary length. For conformance testing this seems reasonable: testing can only be done for a restricted period of time, hence all practical test cases are finite.

How long to test a recursive process with infinite behaviour for conformance is a question of *test selection* too. We refer to it as *vertical test selection*. An item of interest is when testing for infinite, recursive behaviour becomes *reliability testing*, which is defined as testing whether a correct implementation continues to operate correctly after a certain period of time (section 1.1.1).

Image-infiniteness poses more problems in test derivation. Image-infiniteness means that S **after** σ is not finite, in particular that S **after** ϵ is not finite. One way to introduce it in a language is by allowing internal actions to be combined with variables:

$$\mathbf{i}?x : p; B \tag{4.8}$$

with semantics

$$\Sigma \{ \mathbf{i}; B[v/x] \mid p[v/x] \} \tag{4.9}$$

For each possible value v of x: $\mathbf{i}?x : p; B \xrightarrow{\tau} B[v/x]$. If there are infinitely many values v it follows that the acceptance set C is infinite, contrary to $\mathcal{BEX}_v$, where only elements of acceptance sets can be infinite and acceptance sets themselves are always finite. Test derivation for image-infinite processes needs further study.

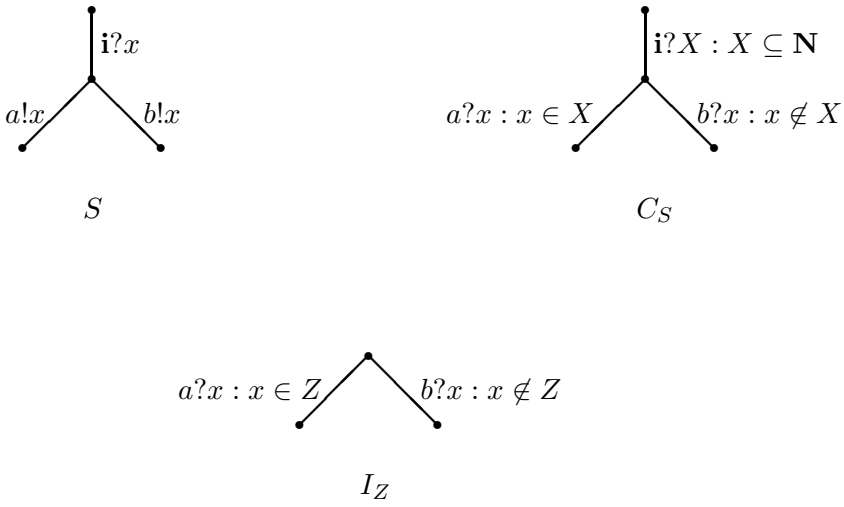


Figure 4.5: Canonical tester for image-infinite process.

In section 4.4 it was noticed that the canonical tester of an expression in $\mathcal{BEX}_v$ cannot be expressed in $\mathcal{BEX}_v$: the canonical tester is image-infinite, while expressions in $\mathcal{BEX}_v$ are not. It would be interesting to investigate whether canonical testers for processes in a language that is extended with image-infiniteness according to (4.8), can be expressed in that language.

Consider S and C_S represented pictorially in figure 4.5:

$$S = \mathbf{i}?x;\ (\ a!x; \mathbf{stop} \ \Box\ b!x; \mathbf{stop}\ )$$

It can be expressed in $\mathcal{BEX}_v$ extended with (4.8). C_S is a canonical tester, and could be expressed as

$$C_S\ =\ \mathbf{i}?X : X \subseteq \mathbf{N};\ (\ a?x : x \in X; \mathbf{stop} \ \Box\ b?x : x \notin X; \mathbf{stop}\ )$$

but this cannot be expressed in $\mathcal{BEX}_v$ with (4.8), since variables with values in $\mathcal{P}(\mathbf{N})$ are not allowed. C_S has uncountably many τ-transitions, while expressions in $\mathcal{BEX}_v$ with (4.8) only have countably many τ-transitions.

Now consider the class of erroneous implementations I_Z with $Z \subset \mathbf{N}$ (figure 4.5). There are uncountably many of such erroneous implementations. They are detected in the canonical tester by the τ-branch with $X = \mathbf{N} \backslash Z$. Moreover, this is the only τ-branch

that can detect I_Z. It follows that there must be uncountably many τ-branches in the canonical tester, and that the canonical tester cannot be expressed in $\mathcal{BEX}_v$ with (4.8).

A possibility of circumventing image-infiniteness for the purpose of conformance testing was already discussed in section 2.2.9 (item 2): it can be required that the PIXIT supplies sufficient information to remove image-infiniteness from a specification for the purpose of test derivation.

Chapter 5

Asynchronous Testing

5.1 Introduction

There are different ways of applying tests to an implementation depending on the test architecture. An important aspect is the nature of the communication between the tester and the implementation. Different ways of interaction between tester and implementation are possible:

- the tester and the protocol implementation can be located in the same computer system. Interactions are implemented by e.g. procedure calls (figure 5.1(a));
- the tester and the protocol implementation can be located in the same computer system and interactions are implemented by putting messages in a buffer (figure 5.1(b));
- the tester is located in another computer system than the protocol implementation, e.g. when the remote test method is used (section 1.3.3, [ISO91a]): interactions take place via the underlying service connecting the tester and the implementation (figure 5.1(c)).

The way the tester is connected to the implementation influences the kind of observations that a tester can make of the implementation. Intuitively, the possibility of making distinct observations decreases when the 'distance' between the tester and the implementation increases. In this chapter we investigate the influence of the kind of communication between the tester and the implementation on testing and test generation. In particular the difference between synchronous communication and asynchronous communication is investigated.

Synchronous communication is characterized by the fact that the communicating partners are involved in an interaction at the same time. An example of this is shaking hands. In synchronous communication there is no real notion of sending and receiving, only of interaction. In the first example above the communication is synchronous (figure 5.1(a), see also figure 2.1).

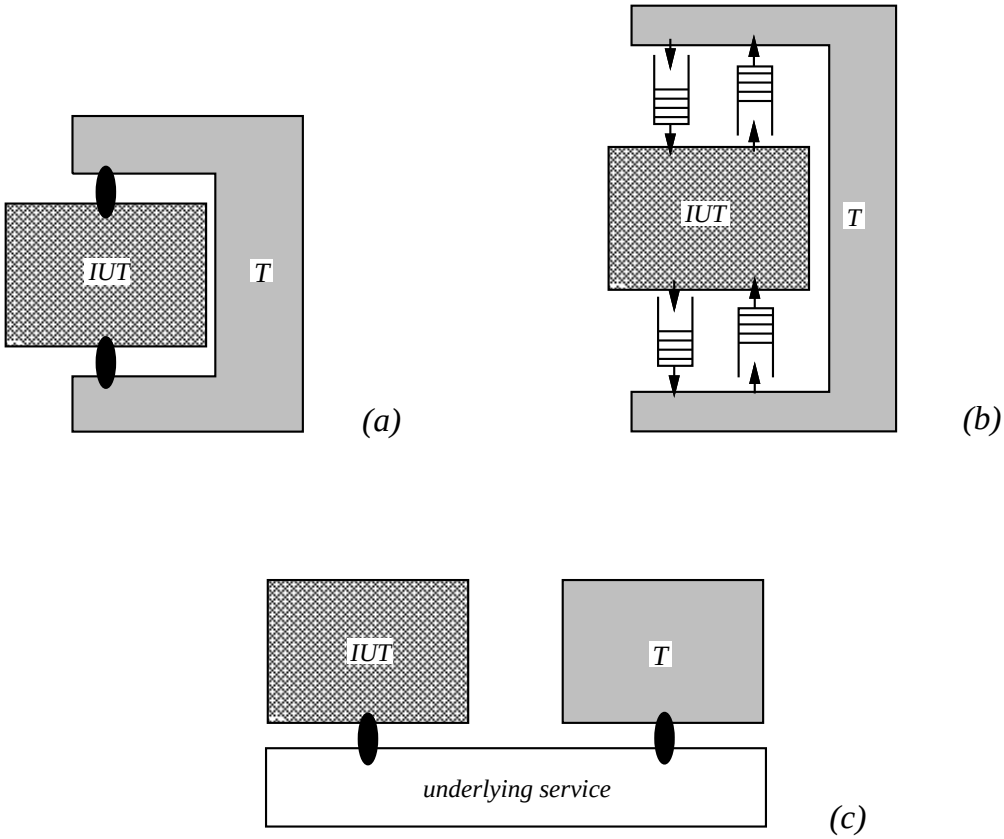


Figure 5.1: Interactions between tester and implementation.

In asynchronous communication the partners perceive the occurrence of communication at different moments in time. The partner that starts the communication is called the sender, the other the receiver. The system only communicates indirectly via some kind of medium with its environment. An example is writing a letter: the moments of writing and reading do not coincide; the postal service constitutes the medium. In section 2.2.6 a *test interface* or *test context* was introduced to model the indirect communication between the implementation and the tester. The test context hides the implementation from its environment. The second and third example above are examples of asynchronous communication (figures 5.1(b) and 5.1(c), cf. figure 2.2).

The theory of testing, conforming implementations, and test generation for labelled transition systems in chapters 3 and 4 was developed assuming synchronous communication between tester and implementation. Synchronous communication is formalized by the synchronous parallel operator $\|$ on labelled transition systems. Testing equivalence $\approx_{te}$ is based on the notion of observation: two labelled transition systems are testing equivalent if there is no (synchronously communicating) external observer that can tell them apart. Choosing a suitable formalization of this intuition of equivalence was shown to lead to the intensional characterization of $\approx_{te}$ in theorem 3.6.

This chapter studies asynchronous communication in the realm of labelled transition systems. Asynchronous communication is simulated by synchronous communication by explicitly introducing a context in the labelled transition system. The context discussed in this chapter consists of a pair of queues, one for input and one for output. The queue context is introduced using a special queue operator on labelled transition systems. Our goal is to explore testing equivalence of specifications in such queue contexts analogous to the approach in chapter 3. We investigate when two specifications that communicate with their environment via queues cannot be distinguished by any environment. Also implementation relations that express correctness of implementations with respect to their specifications, are investigated. It will be shown that test cases generated using the algorithms for synchronous communication (chapter 4) cannot be used for queue contexts. Alternative methods for generating test cases for the queue implementation relations are presented. The relation between the newly defined queue relations and synchronous testing equivalence and implementation relations is given.

Apart from modelling an explicit test context, the theory of queue contexts is applicable to formal languages that have queues built in in their formalism, like the standardized Formal Description Techniques Estelle [ISO89a] and SDL [CCI88], and the standardized test notation TTCN [ISO91a, part 3]. Such formalisms are usually based on finite state machines, communicating via queues. Sending is modelled by putting a message in the input queue of the receiver. Receiving corresponds to taking the first message from the input queue. To apply the labelled transition system based testing theory to these formalisms, or to relate test cases derived from such formalisms to test cases derived from synchronous formalisms, it is necessary to relate their models of communication. The queue model presented in this chapter can serve as a starting point.

The next section starts with an intuitive sketch of the problems encountered when test cases intended for synchronous testing are used for asynchronous testing. In section 5.3

asynchronous communication is formalized by defining the queue operator. Queue equivalence is defined as the finest relation that can be observed if the systems under test communicate with their environment via unbounded queues. The nature of queue equivalence is investigated and an intensional characterization is given in section 5.4. Section 5.5 explores the traces of queue contexts; section 5.6 explores their deadlock behaviour. Implementation relations for queue contexts are considered in section 5.7. Sections 5.8 and 5.9 study test derivation from specifications for asynchronous testing. Finally, some open problems and future work are identified in section 5.10.

5.2 Synchronous versus Asynchronous Testing

Synchronous communication means that interactions between the tester and the implementation under test can only occur if both are prepared to participate in that interaction; otherwise nothing can happen. Formally this means that if the tester T performs an observable action, then also the implementation I must perform the same observable action, and vice versa. This was formalized in chapter 3 using the synchronization operator $\|$ on labelled transition systems.

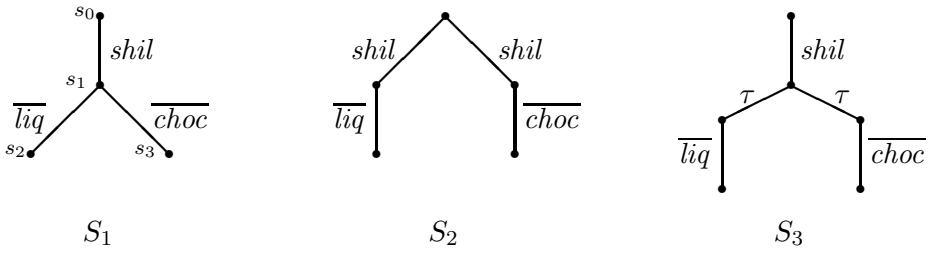


Figure 5.2: Candy machines.

Example 5.1
Figure 5.2 gives examples of labelled transition systems modelling candy machines. Candy machines supply candy via drawers if money is inserted. If the right coins are inserted, drawers will be unlocked so that candy can be obtained. The communication between the machine and a user is synchronous: the action of obtaining candy only occurs if both the drawer is unlocked, and the user pulls a drawer.

Consider the systems of figure 5.2 as implementations, synchronously tested using the test cases of figure 5.3. The test cases are given in $\mathcal{DLTS}$ (section 3.4.3, definition 3.25).

If S_1 is tested with T_1, the result is that S_1 **passes**$_{\mathcal{D}}$ T_1, since the only test run is

$$t_0 \,\|\, s_0 \xrightarrow{shil} t_1 \,\|\, s_1 \xrightarrow{\overline{liq}} t_2 \,\|\, s_2 \;,\quad t_2 \,\|\, s_2 \stackrel{a}{\Longrightarrow} \text{ for all } a \in L \;,\quad \text{and } v(t_2) = \mathbf{pass}$$

If S_2 is tested with T_1 we have the following test runs:

$$T_1 \,\|\, S_2 \stackrel{shil\cdot\overline{liq}}{\Longrightarrow} t_2 \,\|\, S_2', \text{ such that } t_2 \,\|\, S_2' \stackrel{a}{\not\Longrightarrow} \text{ for all } a \in L \text{ , and } v(t_2) = \mathbf{pass}$$
$$T_1 \,\|\, S_2 \stackrel{shil}{\Longrightarrow} t_1 \,\|\, S_2'', \text{ such that } t_1 \,\|\, S_2' \stackrel{a}{\not\Longrightarrow} \text{ for all } a \in L \text{ , and } v(t_1) = \mathbf{fail}$$

hence S_2 $\mathbf{pass\not es}_{\mathcal{D}}$ T_1.

Using T_1 the implementations S_1 and S_2 can be distinguished. The same experiment applied to S_2 and S_3 will show that S_2 and S_3 cannot be distinguished by T_1: S_2, S_3 $\mathbf{pass\not es}_{\mathcal{D}}$ T_1. In fact there is no test case that can distinguish between S_2 and S_3: they are testing equivalent, $S_2 \approx_{te} S_3$.

Now consider S_1 as a specification. T_1 can be obtained using one of the algorithms in chapter 4: it is a test case derived from S_1 for **conf**-testing. Any correct implementation of S_1 must also pass T_1. S_2 $\mathbf{pass\not es}_{\mathcal{D}}$ T_1, hence S_2 is not a correct implementation of S_1: S_2 $\mathbf{con\not f}$ S_1. A complete **conf**-test suite for S_1 is $\{T_1, T_2\}$. It is easy to check that also S_3 $\mathbf{pass\not es}_{\mathcal{D}}$ $\{T_1, T_2\}$.

□

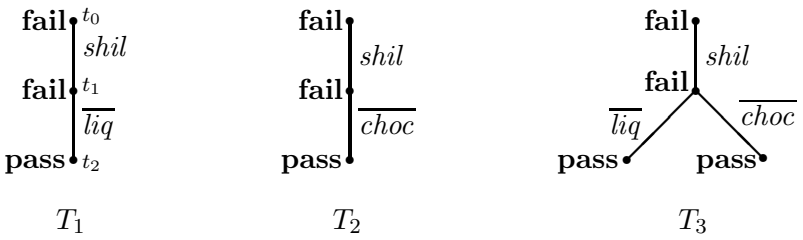


Figure 5.3: Test cases for candy machines.

If the tester and the implementation communicate via a test context, e.g. if they are situated in different computer systems as in the remote test method (figure 5.1(c)), the communication is asynchronous. The occurrence of communication is perceived by the communicating partners at different moments in time.

Example 5.2
Consider again the candy machines (figure 5.2), but now in an asynchronous testing environment. The candy machine cannot be accessed directly; it is connected to two tubes, one to the money slot and one that collects candy from an unlocked drawer. The (remote) tester only has access to the other ends of the tubes. Both tubes have unbounded capacity and pass their contents to the other end in a reliable and order-preserving way. Whenever the machine has a choice between actions it chooses nondeterministically, either taking money from the money tube (money should be available then) or putting candy from an unlocked drawer into the candy tube. The pair of tubes constitutes a test context via which the candy machine is tested. Insertion of a *shilling* is input for the candy machine; supplying candy is output, indicated by a bar: $\overline{choc}$ and $\overline{liq}$.

In this setup the control that the tester has over the machine has diminished considerably, hence also the distinctions that can be observed. Consider again the tests of example 5.1: S_1 is tested with T_1. The tester puts a shilling into the money tube, which is accepted by S_1. S_1 unlocks both the *liquorice*-drawer and the *chocolate*-drawer and chooses nondeterministically which one to put into the candy tube. The tester receives either *liquorice* or *chocolate*. If the tester receives *liquorice* the test is passed, otherwise it fails. Hence, S_1 does not pass T_1 (note that it did using synchronous testing). Analogous reasoning shows that S_2 does not pass T_1. The same thing happens with test cases T_2 and T_3: S_1 and S_2 both fail T_2 and they both pass T_3. In fact no test case can be found that distinguishes between S_1 and S_2: they are testing equivalent for asynchronous testing, but not for synchronous testing.

Also, synchronous test cases cannot be used for asynchronous testing. Consider again S_1 as a specification. Since S_1 does not pass T_1, T_1 is not in any asynchronous test suite derived from S_1. If it were, S_1 itself would not be correct; the corresponding implementation relation would not be reflexive. We saw in example 5.1 that T_1 can be in a synchronous test suite of S_1.

□

Example 5.2 shows that systems can be testing equivalent for asynchronous testing, while they are not testing equivalent for synchronous testing. Moreover, test cases used for synchronous testing (e.g. for **conf**) can produce wrong results when used for asynchronous testing. Other notions of equivalence and conformance, and another way of generating test cases are needed to cope with asynchronous testing. To do this we first need to formalize asynchronous communication, i.e. express it formally in terms of labelled transition systems. This is done in the next section.

5.3 Queue Contexts

Asynchronous communication between two systems, e.g. between an implementation and a tester, can be described by introducing a context. We study a special kind of context, a *queue context*, and we investigate testing equivalence of queue contexts. First, a context relation is defined, then a family of queue operators is introduced to model queue contexts, and then these definitions are combined to define queue equivalence as testing equivalence in a queue context.

Definition 5.3
The *context relation* $\mathcal{R}_{\mathcal{C}} \subseteq \mathcal{LTS} \times \mathcal{LTS}$ with respect to a context $\mathcal{C}[\,\cdot\,] : \mathcal{LTS} \rightarrow \mathcal{LTS}$ and a relation $\mathcal{R} \subseteq \mathcal{LTS} \times \mathcal{LTS}$ is defined by

$$S_1 \;\mathcal{R}_{\mathcal{C}}\; S_2 \quad =_{def} \quad \mathcal{C}[S_1] \;\mathcal{R}\; \mathcal{C}[S_2]$$

□

A context relation relates specifications by relating the behaviour of a given context containing these specifications. We restrict ourselves to contexts consisting of two

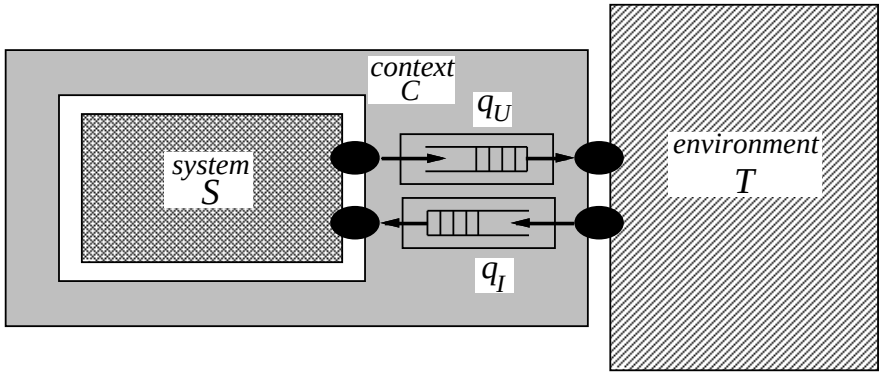


Figure 5.4: Asynchronous communication via unbounded queues.

queues, one queue for input of messages from the environment to the specification, and one queue for output of messages from the specification to the environment. This is depicted in figure 5.4. The system S in figure 5.4 communicates asynchronously with its test environment T by means of an input queue q_I and an output queue q_U. S can put messages in q_U and receive messages from q_I; T can put messages in q_I and receive messages from q_U. q_I and q_U are assumed to be unbounded: they always contain a finite, but arbitrarily large number of messages. Putting messages into the queues and receiving messages from the queues is modelled by events, i.e. occurrences of actions in the label set L. We assume that we can distinguish between input actions L_I and output actions L_U: $L = L_I \cup L_U$, and $L_I \cap L_U = \emptyset$. We use the convention that $a, b, c, \ldots \in L_I$, and $z, y, x, \ldots \in L_U$.

Definition 5.4
A *queue operator* is a unary operator $[_{\sigma_u}\!\ll \cdot \ll_{\sigma_i}] : \mathcal{LTS} \to \mathcal{LTS}$, where $\sigma_i \in L_I^*$ and $\sigma_u \in L_U^*$. Let $S \in \mathcal{LTS}$, then $[_{\sigma_u}\!\ll S \ll_{\sigma_i}] \in \mathcal{LTS}$ is defined by the axioms $A1_Q$ and $A2_Q$ and the inference rules $I1_Q$, $I2_Q$ and $I3_Q$:

$$[_{\sigma_u}\!\ll S \ll_{\sigma_i}] \xrightarrow{a} [_{\sigma_u}\!\ll S \ll_{\sigma_i \cdot a}] \;, \quad a \in L_I \qquad (A1_Q)$$

$$[_{x \cdot \sigma_u}\!\ll S \ll_{\sigma_i}] \xrightarrow{x} [_{\sigma_u}\!\ll S \ll_{\sigma_i}] \;, \quad x \in L_U \qquad (A2_Q)$$

$$\frac{S \xrightarrow{\tau} S'}{[_{\sigma_u}\!\ll S \ll_{\sigma_i}] \xrightarrow{\tau} [_{\sigma_u}\!\ll S' \ll_{\sigma_i}]} \qquad (I1_Q)$$

$$\frac{S \xrightarrow{a} S'}{[_{\sigma_u}\!\ll S \ll_{a \cdot \sigma_i}] \xrightarrow{\tau} [_{\sigma_u}\!\ll S' \ll_{\sigma_i}]} \;, \quad a \in L_I \qquad (I2_Q)$$

$$\frac{S \xrightarrow{x} S'}{[_{\sigma_u}\!\ll S \ll_{\sigma_i}] \xrightarrow{\tau} [_{\sigma_u \cdot x}\!\ll S' \ll_{\sigma_i}]} \;, \quad x \in L_U \qquad (I3_Q)$$

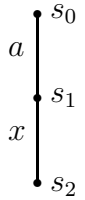
A process Q is a *queue context* if it has the form $[_{\sigma_u}\!\ll S \ll_{\sigma_i}]$ for some S, σ_i, σ_u. The *initial* queue context containing a system S is denoted by Q_S. It is defined as $Q_S =_{def}$

$[_{\epsilon\ll}S_{\ll\epsilon}]$.

□

Example 5.5
Consider the labelled transition system S, with $L_I = \{a, b\}$ and $L_U = \{x, y\}$. The view of an asynchronously communicating observer on S can be expressed as $Q_S = [_{\epsilon\ll}S_{\ll\epsilon}]$. A possible sequence of transitions of Q_S is:

S:



$$\begin{array}{rcll} Q_S & = & [_{\epsilon\ll}s_{0\ll\epsilon}] & \xrightarrow{a} \quad (*\ A1_Q\ *) \\ & & [_{\epsilon\ll}s_{0\ll a}] & \xrightarrow{b} \quad (*\ A1_Q\ *) \\ & & [_{\epsilon\ll}s_{0\ll a\cdot b}] & \xrightarrow{\tau} \quad (*\ I2_Q\ *) \\ & & [_{\epsilon\ll}s_{1\ll b}] & \xrightarrow{\tau} \quad (*\ I3_Q\ *) \\ & & [_{x\ll}s_{2\ll b}] & \xrightarrow{x} \quad (*\ A2_Q\ *) \\ & & [_{\epsilon\ll}s_{2\ll b}] & \end{array}$$

With only observable actions: $[_{\epsilon\ll}s_{0\ll\epsilon}] \stackrel{a\cdot b\cdot x}{\Longrightarrow} [_{\epsilon\ll}s_{2\ll b}]$ Note that $a\cdot b\cdot x$ is not a trace of S. Apparently $[_{\epsilon\ll}S_{\ll\epsilon}]$ can perform traces that S cannot perform.

□

Example 5.6
A formalization of example 5.2 can now be given:

$$t_0 \,\|\, [_{\epsilon\ll}s_{0\ll\epsilon}] \xrightarrow{shil} t_1 \,\|\, [_{\epsilon\ll}s_{0\ll shil}] \xrightarrow{\tau} t_1 \,\|\, [_{\epsilon\ll}s_{1\ll\epsilon}] \xrightarrow{\tau} t_1 \,\|\, [_{\overline{liq}\ll}s_{2\ll\epsilon}] \xrightarrow{\overline{liq}} t_2 \,\|\, [_{\epsilon\ll}s_{2\ll\epsilon}]\ ,$$
$$\text{with}\ \ t_2 \,\|\, [_{\epsilon\ll}s_{2\ll\epsilon}] \stackrel{a}{\not\Longrightarrow}\ \ \text{for all } a \in L\ \ ,\ \ \text{and}\ \ v(t_2) = \mathbf{pass}$$

and

$$t_0 \,\|\, [_{\epsilon\ll}s_{0\ll\epsilon}] \xrightarrow{shil} t_1 \,\|\, [_{\epsilon\ll}s_{0\ll shil}] \xrightarrow{\tau} t_1 \,\|\, [_{\epsilon\ll}s_{1\ll\epsilon}] \xrightarrow{\tau} t_1 \,\|\, [_{\overline{choc}\ll}s_{3\ll\epsilon}]\ ,$$
$$\text{with}\ \ t_1 \,\|\, [_{\overline{choc}\ll}s_{3\ll\epsilon}] \stackrel{a}{\not\Longrightarrow}\ \ \text{for all } a \in L\ \ ,\ \ \text{and}\ \ v(t_1) = \mathbf{fail}$$

hence Q_{S_1} $\mathbf{passes}_{\mathcal{D}}$ T_1. An analogous derivation can be given for Q_{S_2} $\mathbf{passes}_{\mathcal{D}}$ T_1.

Q_{S_1} $\mathbf{passes}_{\mathcal{D}}$ T_3:

$$T_3 \,\|\, [_{\epsilon\ll}S_{1\ll\epsilon}] \stackrel{shil\cdot\overline{liq}}{\Longrightarrow} T_3' \,\|\, [_{\epsilon\ll}s_{2\ll\epsilon}] \stackrel{a}{\not\Longrightarrow}\ \ \text{for all } a \in L\ \ ,\ \ \text{and}\ \ v(T_3') = \mathbf{pass}$$
$$T_3 \,\|\, [_{\epsilon\ll}S_{1\ll\epsilon}] \stackrel{shil\cdot\overline{choc}}{\Longrightarrow} T_3'' \,\|\, [_{\epsilon\ll}s_{3\ll\epsilon}] \stackrel{a}{\not\Longrightarrow}\ \ \text{for all } a \in L\ \ ,\ \ \text{and}\ \ v(T_3'') = \mathbf{pass}$$

□

The queue context $[_{\epsilon\ll}S_{\ll\epsilon}]$ models the asynchronous communication between a system S and its environment. The queue context itself is a labelled transition system that communicates synchronously with its environment. This means that observing S asynchronously corresponds to observing $[_{\epsilon\ll}S_{\ll\epsilon}]$ synchronously. Hence, we should apply the theory of synchronous communication presented in chapter 3 to $[_{\epsilon\ll}S_{\ll\epsilon}]$, which implies that the finest distinctions that can be made between queue contexts are given by testing equivalence $\approx_{te}$. The equivalence on the specifications contained in the queue contexts induced in this way is called *queue equivalence*: it is the context equivalence with respect to $[_{\epsilon\ll}\cdot_{\ll\epsilon}]$ and $\approx_{te}$.

Definition 5.7
Queue equivalence $\approx_{\mathcal{Q}} \subseteq \mathcal{LTS} \times \mathcal{LTS}$ is defined by

$$S_1 \approx_{\mathcal{Q}} S_2 \quad =_{def} \quad [_{\epsilon} \ll S_1 \ll_{\epsilon}] \approx_{te} [_{\epsilon} \ll S_2 \ll_{\epsilon}]$$

□

Our objective is to explore the nature of $\approx_{\mathcal{Q}}$ and to relate it to other equivalences on $\mathcal{LTS}$. Before starting with a formal treatment of queue equivalence in the next sections, we study a few examples to get some intuition about the kind of distinctions that can be made and that cannot be made.

The first thing to observe is that a queue context *always* yields a labelled transition system with infinite behaviour (if $L_I \neq \emptyset$), because it can always accept any number of input actions, due to axiom $A1_Q$. From this observation we can also conclude that systems which do not produce any outputs can never be distinguished.

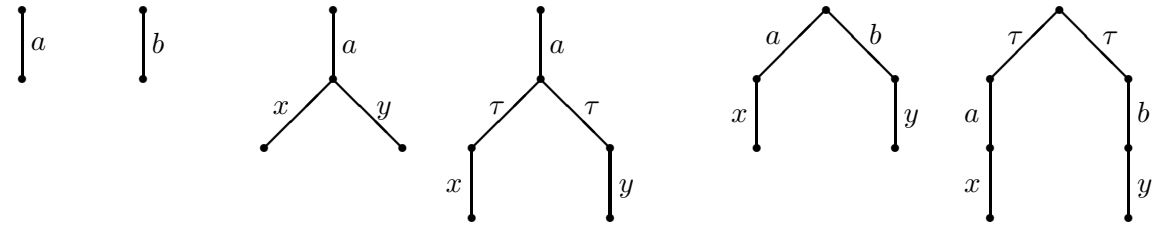


Figure 5.5: Examples for asynchronous observation.

Example 5.8
Consider figure 5.5. Q_{S_1} and Q_{S_2} cannot be distinguished by any environment. Apparently, a system must produce output actions in order to make any relevant observations. Also the processes Q_{T_1} and Q_{T_2} cannot be distinguished. An output is nondeterministically generated by the system. No environment can have any influence on that. The reason is inference rule $I3_Q$ which introduces a τ-step preceding any output action. Q_{U_1} and Q_{U_2} can be distinguished, e.g. by an environment trying to do $a \cdot x$. With Q_{U_1} it will always succeed; with Q_{U_2} it may deadlock after having done a. This implies that nondeterministic choice between input actions can be observed, whereas Q_{T_1} and Q_{T_2} show that it cannot be observed for output actions.

□

The first example shows that queue equivalence does not imply trace equivalence $\approx_{tr}$, and thus not testing equivalence $\approx_{te}$ either. The second example shows that also in the case that the traces are equal $\approx_{\mathcal{Q}}$ does not imply $\approx_{te}$. The third example shows that $\approx_{\mathcal{Q}}$ is not implied by trace equivalence either.

Example 5.9
Consider figure 5.6. A system S is given, and (a testing equivalent version of) its

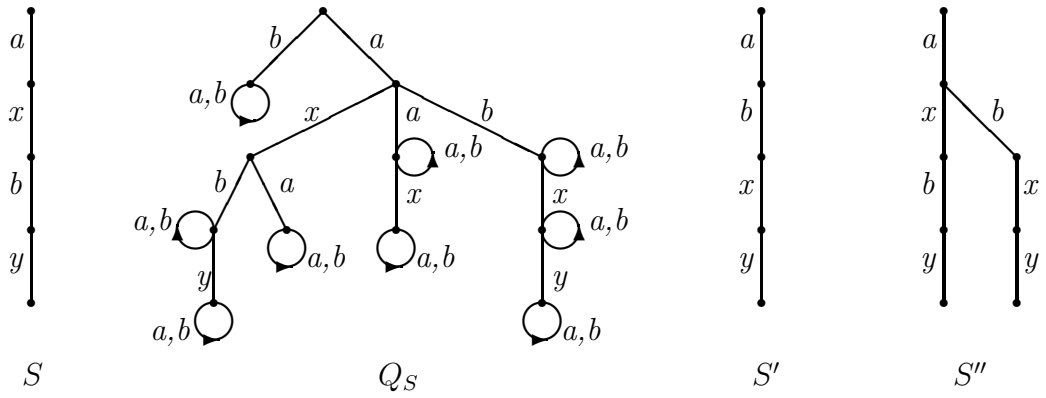


Figure 5.6: An example of a queue context.

queue context Q_S, where τ-steps have been removed. First we see that Q_S has infinite behaviour: e.g. $a{\cdot}x{\cdot}a{\cdot}a{\cdot}a\ldots$ is a trace of Q_S. Secondly, we see that $a{\cdot}x{\cdot}b{\cdot}y$ is a trace of S and of Q_S. $a{\cdot}b{\cdot}x{\cdot}y$ is a trace of Q_S but not of S. Apparently we can 'shift' input actions to the front of the trace to get a trace of Q_S that is not a trace of S. The reason for this phenomenon is that because of axiom $A1_Q$ we can do input actions 'in advance', which remain in the queue, before doing the output actions. While Q_S is doing $a{\cdot}b{\cdot}x{\cdot}y$, S is doing $a{\cdot}x{\cdot}b{\cdot}y$. Something similar can be observed in some operating systems which allow impatient users to type in their commands in advance while the program is still busy working. The other way around, i.e. shifting input actions to the end, does not yield a trace of Q_S: $x{\cdot}a{\cdot}b{\cdot}y$, nor $a{\cdot}x{\cdot}y{\cdot}b$ is a trace of Q_S.

□

Example 5.10
One might suspect that shifting input actions to the front yields queue equivalent specifications. This is only the case if the original trace is not removed: S' is not queue equivalent to S (try the experiment $a{\cdot}x$), but S'' is (figure 5.6).

□

5.4 Queue Equivalence

Queue equivalence $\approx_{\mathcal{Q}}$ was defined (definition 5.7) as testing equivalence $\approx_{te}$ (theorem 3.6) in a queue context (definition 5.4):

$$\begin{array}{ll} & S_1 \approx_{\mathcal{Q}} S_2 \\ \text{iff} & Q_{S_1} \approx_{te} Q_{S_2} \\ \text{iff} & \forall \sigma \in L^*, \forall A \subseteq L:\ Q_{S_1} \textbf{ after } \sigma \textbf{ must } A \ \text{ iff } \ Q_{S_2} \textbf{ after } \sigma \textbf{ must } A \end{array} \tag{5.1}$$

The specific properties of queue contexts that came up in examples 5.8, 5.9 and 5.10 allow us to derive simpler characterizations of $\approx_{\mathcal{Q}}$. These properties are elaborated leading to theorem 5.14 and corollary 5.15.

First consider the input actions of a queue context Q. According to axiom $A1_Q$ (definition 5.4) there is no restriction in performing an input action $a \in L_I$: any queue context can always perform any input action. Consequently, Q **after** σ **must** A always holds, if A contains an input action:

Proposition 5.11

1. Any queue context Q can always do any sequence of input actions: $\forall \sigma_i \in L_I^* : Q \stackrel{\sigma_i}{\Longrightarrow}$
2. For a queue context Q, $\sigma \in L^*$, $A \subseteq L$, and $A \cap L_I \neq \emptyset$, Q **after** σ **must** A holds.

□

Now consider the output actions. In example 5.8 we saw that output actions of a queue context always occur nondeterministically. This is due to the internal τ-step which occurs when the system S puts an output action x into the output queue. After this τ-step the output action x is contained in the output queue, and the environment has to remove x before it can perform any subsequent output action. This can be expressed as follows: if for some Q': $Q \stackrel{\sigma}{\Longrightarrow} Q' \stackrel{x}{\longrightarrow}$, then for all $y \neq x$: $Q \stackrel{\sigma}{\Longrightarrow} Q' \stackrel{y}{\not\Longrightarrow}$, which means: Q **after** σ **refuses** $L_U \backslash \{x\}$:

Proposition 5.12
If $Q \stackrel{\sigma \cdot x}{\Longrightarrow}$ then Q **after** σ **refuses** $L_U \backslash \{x\}$.

□

The nondeterministic occurrence of output actions corresponds to the intuition of sending and receiving: the environment can receive the actions sent by S, but it does not have any direct influence on which actions will be sent. The specification of S determines which output actions *may* be produced. It *may* also occur that no output actions are produced at all: Q **after** σ **refuses** L_U ; the system is in an *output deadlock*. This discussion suggests that the complete behaviour of a queue context is determined by the output actions that may occur after a certain trace, with the possibility that no output action occurs. Theorem 5.14 shows that this is correct.

Definition 5.13

1. the *output function* $outputs_S : L^* \rightarrow \mathcal{P}(L_U)$, is defined by

$$outputs_S(\sigma) \quad =_{def} \quad \{x \in L_U \mid Q_S \stackrel{\sigma \cdot x}{\Longrightarrow}\}$$

2. the *output deadlock predicate* δ_S over L^*, is defined by

$$\delta_S(\sigma) \quad =_{def} \quad Q_S \textbf{ after } \sigma \textbf{ refuses } L_U$$

The traces leading to output deadlock are denoted by

$$\delta\text{-}traces(S) \quad =_{def} \quad \{\sigma \in L^* \mid \delta_S(\sigma)\}$$

3. the *observation function* $\mathcal{O}_S : L^* \rightarrow \mathcal{P}(L_U \cup \{\delta\})$, $\delta \notin L$, is defined by

$$\mathcal{O}_S(\sigma) \quad =_{def} \quad \begin{cases} outputs_S(\sigma) & \text{if } \neg\delta_S(\sigma) \\ outputs_S(\sigma) \cup \{\delta\} & \text{if } \delta_S(\sigma) \end{cases}$$

□

The *observation function* $\mathcal{O}_S(\sigma)$ gives all output actions that *may* occur after σ. The special symbol $\delta \notin L$ is used to indicate that an output deadlock *may* occur after σ. It is necessary in order to distinguish between the possibility of Q_S not being able to produce any output *after* σ ($\delta \in \mathcal{O}_S(\sigma)$), and Q_S not being able to do σ ($\sigma \notin traces(Q_S)$ iff $\mathcal{O}_S(\sigma) = \emptyset$).

Theorem 5.14

$$S_1 \approx_{\mathcal{Q}} S_2 \quad \text{iff} \quad \forall \sigma \in L^* : \mathcal{O}_{S_1}(\sigma) = \mathcal{O}_{S_2}(\sigma)$$

□

Theorem 5.14 states that two processes are queue equivalent if and only if their observations after any trace of actions are equal, where an observation is the occurrence of an output action, or the observation that the output queue is empty. Theorem 5.14 can be rewritten, using the definition of $\mathcal{O}_S(\sigma)$ and proposition 5.11.1, to equality of two sets of traces: the normal traces of Q_S, and the traces of Q_S that lead to an output deadlock. These traces were defined as the δ-*traces* of S.

Corollary 5.15

$$S_1 \approx_{\mathcal{Q}} S_2 \quad \text{iff} \quad traces(Q_{S_1}) = traces(Q_{S_2}) \text{ and } \delta\text{-}traces(S_1) = \delta\text{-}traces(S_2)$$

□

Example 5.16
Consider again figure 5.5. For S_1 and S_2, for all $\sigma_i \in L_I^*$: $\mathcal{O}_{S_1}(\sigma_i) = \mathcal{O}_{S_1}(\sigma_i) = \{\delta\}$, while for all $\sigma \notin L_I^*$: $\mathcal{O}_{S_1}(\sigma) = \mathcal{O}_{S_1}(\sigma) = \emptyset$. $\mathcal{O}_{T_1}(a) = \mathcal{O}_{T_2}(a) = \{x, y\}$, $\mathcal{O}_{T_1}(\epsilon) = \mathcal{O}_{T_2}(\epsilon) = \mathcal{O}_{T_1}(a \cdot x) = \mathcal{O}_{T_2}(a \cdot x) = \{\delta\}$. The processes U_1 and U_2 are not equivalent: $\mathcal{O}_{U_1}(a) = \{x\} \neq \{x, \delta\} = \mathcal{O}_{U_2}(a)$.
□

Processes modulo queue equivalence $\approx_{\mathcal{Q}}$ are completely characterized by linear structures (i.e. traces), as opposed to the must sets (or acceptance sets, or failure sets) necessary to characterize processes modulo testing equivalence $\approx_{te}$ (cf. [BKPR91]: 'The failure of failures in a paradigm for asynchronous communication'). This characterization seems easy and straightforward, however, for other than purely mathematical purposes it has a severe drawback: $traces(Q_S)$ and δ-$traces(S)$ are infinite for every process S (if $L_I \neq \emptyset$). The next sections study possibilities of reducing the sizes of the sets $traces(Q_S)$ and δ-$traces(S)$ in order to obtain a finite representation of processes modulo $\approx_{\mathcal{Q}}$, at least for processes with finite behaviour.

5.5 Traces of Queue Contexts

We consider the relation between traces of a process S and its queue context Q_S. In example 5.9 (section 5.3) it was already shown that $traces(S)$ and $traces(Q_S)$ are not equal. Sometimes actions can be shifted with respect to each other. The exact nature of this reordering of actions is studied here.

The relation @ ('ape') is introduced to describe this reordering. If $\sigma_1 @ \sigma_2$ (σ_1 *is aped by* σ_2), then σ_2 is obtained from σ_1 by a combination of shifting input actions to the front and adding inputs at the end of σ_1. This shifting of input actions corresponds to Q_S doing input actions 'in advance': they stay in the input queue until they are consumed by S. We can also say that an observer of Q_S can leave the output actions produced by S in the output queue, thus shifting output actions to the end of the trace that was performed by S. Adding input actions at the end can be explained since Q_S can always do input actions. The operation $\backslash\backslash$ is introduced to produce the sequence of input actions that was added to σ_1.

We show that @ precisely characterizes the relation between traces of S and of Q_S. This is done in corollary 5.23, which is a consequence of propositions 5.20 and 5.22.

The relation @ between σ_1 and σ_2 holds, if the outputs of σ_1 and σ_2 are equal, the inputs of σ_1 are a prefix of the inputs of σ_2, and inputs that precede certain outputs in σ_1, precede the same outputs in σ_2:

Definition 5.17

1. The relation $@ \subseteq L^* \times L^*$ is defined as the smallest relation such that:
   - if $\sigma_1, \sigma_2 \in L_I^*$, then $\sigma_1 @ \sigma_2 \;\; =_{def} \;\; \sigma_1 \preceq \sigma_2$
   - if $\sigma_1 = \rho_1 \cdot x_1 \cdot \sigma_1'$, $\sigma_2 = \rho_2 \cdot x_2 \cdot \sigma_2'$, with $\rho_1, \rho_2 \in L_I^*$, $x_1, x_2 \in L_U$, and $\sigma_1', \sigma_2' \in L^*$, then $\sigma_1 @ \sigma_2 \;\; =_{def} \;\; \rho_1 \preceq \rho_2$ and $x_1 = x_2$ and $\sigma_1' @ (\rho_2 \backslash \rho_1) \cdot \sigma_2'$
2. If $\sigma_1 @ \sigma_2$, then the operation $\backslash\backslash$ is defined by
   - if $\sigma_1, \sigma_2 \in L_I^*$, then $\sigma_2 \backslash\backslash \sigma_1 \;\; =_{def} \;\; \sigma_2 \backslash \sigma_1$
   - if $\sigma_1 = \rho_1 \cdot x_1 \cdot \sigma_1'$, $\sigma_2 = \rho_2 \cdot x_2 \cdot \sigma_2'$, with $\rho_1, \rho_2 \in L_I^*$, $x_1, x_2 \in L_U$, and $\sigma_1', \sigma_2' \in L^*$, then $\sigma_2 \backslash\backslash \sigma_1 \;\; =_{def} \;\; ((\rho_2 \backslash \rho_1) \cdot \sigma_2') \backslash\backslash \sigma_1'$

   Note that $\backslash\backslash$ is well-defined, since $\sigma_1 @ \sigma_2$ implies $\rho_1 \preceq \rho_2$ and $\sigma_1' @ (\rho_2 \backslash \rho_1) \cdot \sigma_2'$. □

Proposition 5.18
Let $\sigma, \sigma_1, \sigma_2 \in L^*$:

1. $\sigma_1 @ \sigma_2$ implies $\sigma_1 \lceil L_U = \sigma_2 \lceil L_U$
2. $\sigma_1 @ \sigma_2$ implies $\sigma_1 \lceil L_I \preceq \sigma_2 \lceil L_I$
3. $\sigma \lceil L_U @ \sigma$
4. $\sigma_2 \backslash\backslash \sigma_1 = (\sigma_2 \lceil L_I) \backslash (\sigma_1 \lceil L_I)$

□

Another way of expressing the relation @ is by identifying the *causal relations* between occurrences of actions in traces: an occurrence of an action e_2 causally depends on e_1 if it is necessary that e_1 occurs before e_2 can occur [Lan92]. $\sigma_1 @ \sigma_2$ if the following causal dependencies among the occurrences of actions in σ_1 are preserved in σ_2:

- causal dependencies of input actions on previous input actions;
- causal dependencies of output actions on previous output actions;
- causal dependencies of output actions on previous input actions.

The causal dependencies of input actions on output actions in σ_1 need not be preserved.

Example 5.19
$x \cdot a \,@\, x \cdot a \cdot b \cdot a$ (adding inputs at the end), $x \cdot a \cdot b \cdot a \backslash\backslash x \cdot a = b \cdot a$, ($b \cdot a$ was added to $x \cdot a$). $x \cdot a \,@\, a \cdot x$ (shifting inputs to the front), $a \cdot x \backslash\backslash x \cdot a = \epsilon$ (nothing was added to $x \cdot a$). $x \cdot a \,@\, a \cdot b \cdot x$ (a combination of addition and shifting), $a \cdot b \cdot x \backslash\backslash x \cdot a = b$.

Not $x \cdot a \,@\, x \cdot a \cdot y$ (adding outputs is not allowed), not $a \cdot x \,@\, x \cdot a$ (shifting inputs to the end is not allowed), nor $x \cdot a \,@\, x \cdot b$ (removing actions is not allowed).

□

The relation @ is a partial order. Moreover, it is well-founded: there is no infinite descending sequence of traces (appendix A). Intuitively this can be seen as follows: let $\sigma' @ \sigma$, then σ' is obtained from σ by shifting input actions to the end and/or removing input actions from the end of σ. This process of shifting and removing ends after a finite number of steps when only output actions are left: $\sigma \lceil L_U$. In this way every descending sequence ends in a trace consisting of only output actions (see also proposition 5.18).

Proposition 5.20
$\langle L^*, @ \rangle$ is a well-founded poset.

□

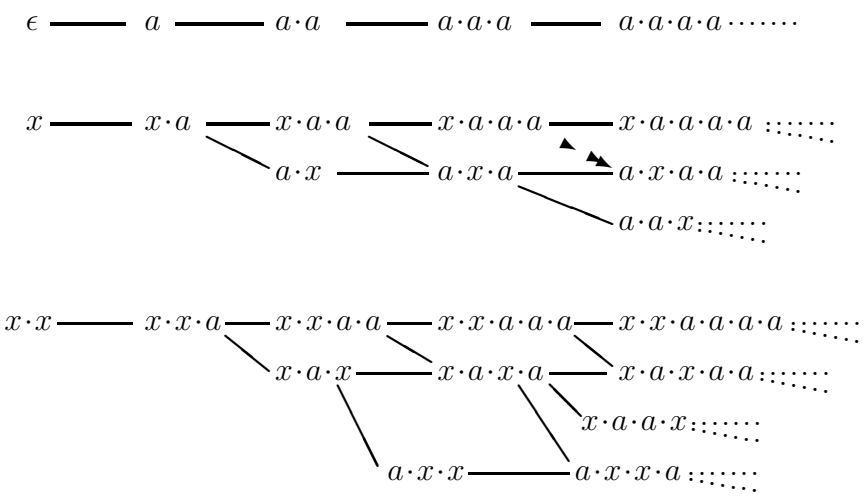


Figure 5.7: @ as partial order over $\{a, x\}^*$

Example 5.21
Consider the order diagram of $\langle L^*, @\rangle$ with $L_I = \{a\}$ and $L_U = \{x\}$ (figure 5.7). The order consists of unrelated sub-orders, with ϵ, x, $x \cdot x$, ..., as minimal elements: $min_@(L^*) = L_U^*$.

□

Now the relation between transitions of a labelled transition system S and its queue context Q_S is given in proposition 5.22. Using this proposition corollary 5.23 shows that @ captures exactly the relation between traces of S and traces of Q_S.

Proposition 5.22

1. If $\sigma_1 @ \sigma_2$ and $S \stackrel{\sigma_1}{\Longrightarrow} S'$ then $[_\epsilon \ll S \ll_\epsilon] \stackrel{\sigma_2}{\Longrightarrow} [_\epsilon \ll S' \ll_{\sigma_2 \backslash\backslash \sigma_1}]$
2. If $[_\epsilon \ll S \ll_\epsilon] \stackrel{\sigma_2}{\Longrightarrow} [_\epsilon \ll S' \ll_{\sigma_r}]$ then $\exists \sigma_1 : S \stackrel{\sigma_1}{\Longrightarrow} S'$, $\sigma_1 @ \sigma_2$, and $\sigma_r = \sigma_2 \backslash\backslash \sigma_1$
3. $Q_S \stackrel{\sigma}{\Longrightarrow}$ iff $\exists S', \sigma_i', \sigma_u' : [_\epsilon \ll S \ll_\epsilon] \stackrel{\sigma}{\Longrightarrow} [_{\sigma_u'} \ll S' \ll_{\sigma_i'}]$
   iff $\exists S'', \sigma_i'' : [_\epsilon \ll S \ll_\epsilon] \stackrel{\sigma}{\Longrightarrow} [_\epsilon \ll S'' \ll_{\sigma_i''}]$

□

Corollary 5.23

1. $\sigma_1 @ \sigma_2$ and $\sigma_1 \in traces(S)$ imply $\sigma_2 \in traces(Q_S)$
2. $\sigma_1 @ \sigma_2$ and $\sigma_1 \in traces(Q_S)$ imply $\sigma_2 \in traces(Q_S)$
3. $\sigma_2 \in traces(Q_S)$ implies $\exists \sigma_1 \in traces(S) : \sigma_1 @ \sigma_2$

□

The trace σ_1 in corollary 5.23.3 is not unique: it can occur that two traces σ_1' and σ_1 with $\sigma_1' @ \sigma_1$ are both traces of S. If we observe the trace σ_2 of Q_S, there is no way to decide whether σ_2 is a consequence of S performing σ_1' or σ_1. We can remove σ_1 from the traces of S without changing the traces of Q_S.

Corollary 5.23.2 expresses that $traces(Q_S)$ is right-closed with respect to @: $traces(Q_S) = \overline{traces(Q_S)}$. Together with the well-foundedness of $\langle L^*, @\rangle$ (proposition 5.20) it follows that proposition A.2 is applicable: $traces(Q_S)$ can be represented by its minimal elements. These minimal elements are called the *tracks* of S.

Corollary 5.23.1 and 5.23.3 express that $traces(Q_S)$ is equal to the right-closure of $traces(S)$. Using proposition A.1 it follows that the minimal elements of $traces(S)$ and of $traces(Q_S)$ are equal. So using tracks $traces(Q_S)$ can be represented by a set, whose size does not exceed that of $traces(S)$. This means that for finite $traces(S)$ we have a finite representation of $traces(Q_S)$.

Definition 5.24
$tracks(S) \quad =_{def} \quad min_@(traces(S))$

□

Proposition 5.25

1. $tracks(S) \subseteq traces(S) \subseteq traces(Q_S) \subseteq L^*$
2. $traces(Q_S) = \overline{traces(S)} = \overline{tracks(S)}$

3. $tracks(S) = min_{@}(traces(Q_S))$
4. A track is either equal to ϵ, or it ends with an output action.

□

Theorem 5.26

$$tracks(S_1) = tracks(S_2) \quad \text{iff} \quad traces(Q_{S_1}) = traces(Q_{S_2})$$

□

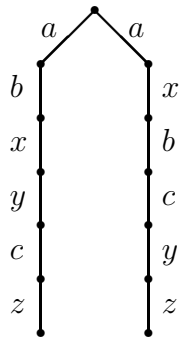


Figure 5.8: Specification S.

Example 5.27
Consider figure 5.6: $tracks(S) = \{\epsilon,\ a{\cdot}x,\ a{\cdot}x{\cdot}b{\cdot}y\}$. For every trace σ of S there is a track σ_t such that $\sigma_t @ \sigma$.

Also in figure 5.6: $tracks(S') = \{\epsilon,\ abx,\ abxy\}$; $tracks(S'') = \{\epsilon,\ ax,\ axby\} = tracks(S)$. $a{\cdot}b{\cdot}x,\ a{\cdot}b{\cdot}x{\cdot}y \in traces(S''),$ but $a{\cdot}b{\cdot}x,\ a{\cdot}b{\cdot}x{\cdot}y \notin tracks(S'')$.

Using proposition 5.25.3 and the definition of right-closedness (appendix A):

$$\sigma \in traces(Q_S) \text{ implies } \exists \sigma' \in tracks(S) : \sigma' @ \sigma$$

which means that every trace of Q_S 'is generated' by a track of S. Note that this σ' is not unique: in figure 5.8 the trace $a{\cdot}b{\cdot}c{\cdot}x{\cdot}y{\cdot}z$ of Q_S can be the result of the track $a{\cdot}b{\cdot}x{\cdot}y{\cdot}c{\cdot}z$ or of $a{\cdot}x{\cdot}b{\cdot}c{\cdot}y{\cdot}z$.

□

5.6 Output Deadlocks of Queue Contexts

In the previous section we saw that the traces of a queue context are completely characterized by the tracks of the specification. Now we investigate the output deadlocks of a queue context. This will lead to a characterization of queue equivalence with three different sets, which are all finite for finite specifications (theorem 5.36). For this characterization we distinguish between temporary output deadlocks and permanent output

deadlocks, depending on the fact whether the deadlock can be resolved or not. To relate output deadlocks of a queue context Q_S to the specification S we distinguish between deadlocks that are caused by an empty input queue, and deadlocks that are caused by blocking of the input queue. This section concludes with relating these four different kinds of output deadlocks.

Temporary and permanent output deadlocks

A queue context is in temporary output deadlock if it waits for inputs: it cannot output anything, but after some suitable inputs it will generate outputs again. In a permanent output deadlock it is possible that no output is generated, regardless what extra inputs are given to the queue context.

Definition 5.28

1. $\delta\text{-}temp(S) \;=_{def}\; \{\, \sigma \in L^* \mid \delta_S(\sigma) \text{ and } \exists \sigma' \in L^* : \sigma \mathbin{@} \sigma' \text{ and } \neg\delta_S(\sigma') \,\}$
2. $\delta\text{-}perm(S) \;=_{def}\; \{\, \sigma \in L^* \mid \delta_S(\sigma) \text{ and } \forall \sigma' \in L^* : \text{if } \sigma \mathbin{@} \sigma' \text{ then } \delta_S(\sigma') \,\}$

□

Proposition 5.29

1. $\delta\text{-}temp(S)$ and $\delta\text{-}perm(S)$ form a partition of $\delta\text{-}traces(S)$
2. $\delta\text{-}traces(S_1) = \delta\text{-}traces(S_2)$
   iff $\delta\text{-}temp(S_1) = \delta\text{-}temp(S_2)$ and $\delta\text{-}perm(S_1) = \delta\text{-}perm(S_2)$

□

In δ-*perm* and in δ-*temp* we can find minimal elements, which completely represent these sets, in a similar way as the tracks of a specification determine the traces of its queue context.

Representing δ-*perm*

For δ-*perm* the situation is analogous to the traces of queue contexts: δ-*perm* is right-closed with respect to @, as follows immediately from definition 5.28, which, combined with the well-foundedness of $\langle L^*, @\rangle$, implies that δ-*perm* is completely characterized by its minimal elements with respect to @. These minimal elements are called *P-tracks*.

Definition 5.30
$P\text{-}tracks(S) \;=_{def}\; min_{@}(\delta\text{-}perm(S))$

□

Proposition 5.31
$\delta\text{-}perm(S_1) = \delta\text{-}perm(S_2)$ iff $P\text{-}tracks(S_1) = P\text{-}tracks(S_2)$

□

Representing δ-*temp*

The set δ-*temp* cannot be represented by its minimal elements with respect to @. We need another partial order: $|@|$. This relation is a restriction of @: $\sigma_1 \,|@|\, \sigma_2$ if σ_2 can be obtained from σ_1 by only shifting inputs to the front; adding input actions at the end of σ_1 is not allowed, hence the length of σ_1 and σ_2 is the same. With this extra restriction variants of propositions 5.22.1 and 5.22.2 can be formulated (propositions 5.33.3 and 5.33.4). Proposition 5.33.5 is a variant of corollary 5.23.2: if $\sigma_1 \,|@|\, \sigma_2$, then not only $Q_S \stackrel{\sigma_1}{\Longrightarrow}$ implies $Q_S \stackrel{\sigma_2}{\Longrightarrow}$, but also the resulting states can be the same. Proposition 5.33.6 shows the importance of $|@|$ for output deadlocks. It is the analogue of corollary 5.23.2 for deadlock traces: δ-*traces* is right-closed with respect to $|@|$.

Definition 5.32
$\sigma_1 \,|@|\, \sigma_2 \;\; =_{def} \;\; \sigma_1 \,@\, \sigma_2$ and $|\sigma_1| = |\sigma_2|$

□

Proposition 5.33

1. $\langle L^*, |@|\,\rangle$ is a well-founded poset.
2. $\sigma_1 \,@\, \sigma_2$ implies $\sigma_1 \cdot (\sigma_2 \backslash\backslash \sigma_1) \,|@|\, \sigma_2$
3. If $\sigma_1 \,|@|\, \sigma_2$ and $S \stackrel{\sigma_1}{\Longrightarrow} S'$ then $[_\epsilon\!\ll\! S \!\ll_\epsilon] \stackrel{\sigma_2}{\Longrightarrow} [_\epsilon\!\ll\! S' \!\ll_\epsilon]$
4. If $[_\epsilon\!\ll\! S \!\ll_\epsilon] \stackrel{\sigma_2}{\Longrightarrow} [_\epsilon\!\ll\! S' \!\ll_\epsilon]$ then $\exists \sigma_1 : S \stackrel{\sigma_1}{\Longrightarrow} S'$ and $\sigma_1 \,|@|\, \sigma_2$
5. If $\sigma_1 \,|@|\, \sigma_2$ and $Q_S \stackrel{\sigma_1}{\Longrightarrow} Q'$ then $Q_S \stackrel{\sigma_2}{\Longrightarrow} Q'$
6. If $\delta_S(\sigma_1)$ and $\sigma_1 \,|@|\, \sigma_2$ then $\delta_S(\sigma_2)$

□

Similar to δ-*perm*, we introduce the *T-tracks* as the minimal elements of δ-*temp* with respect to $|@|$:

Definition 5.34
T-tracks$(S) \;\; =_{def} \;\; min_{|@|}(\delta\text{-}temp(S))$

□

Proposition 5.35
For $S_1, S_2 \in \mathcal{LTS}$ such that *P-tracks*(S_1) = *P-tracks*(S_2):
δ-*temp*$(S_1) = \delta$-*temp*(S_2) iff *T-tracks*(S_1) = *T-tracks*(S_2)

□

An alternative characterization of queue equivalence

With the *tracks*, *P-tracks* and *T-tracks* we have all the ingredients for an alternative characterization of queue equivalence, as follows from corollary 5.15, using theorem 5.26 and propositions 5.29.2, 5.31, and 5.35.

Theorem 5.36

$$S_1 \approx_{\mathcal{Q}} S_2 \quad \text{iff} \quad \begin{array}{ll} \mathit{tracks}(S_1) = \mathit{tracks}(S_2) & \text{and} \\ \mathit{P\text{-}tracks}(S_1) = \mathit{P\text{-}tracks}(S_2) & \text{and} \\ \mathit{T\text{-}tracks}(S_1) = \mathit{T\text{-}tracks}(S_2) & \end{array}$$

□

S_1	S_2	S_3	S_4
$\mathtt{tracks} = \{\epsilon, x\}$	$\mathtt{tracks} = \{\epsilon, x\}$	$\mathtt{tracks} = \{\epsilon,\ b{\cdot}x\}$	$\mathtt{tracks} = \{\epsilon,\ b{\cdot}x\}$
$\mathtt{T\text{-}tracks} = \emptyset$	$\mathtt{T\text{-}tracks} = \emptyset$	$\mathtt{T\text{-}tracks} = \{\epsilon\}$	$\mathtt{T\text{-}tracks} = \{\epsilon\}$
$\mathtt{P\text{-}tracks} = \{a, x\}$	$\mathtt{P\text{-}tracks} = \{x\}$	$\mathtt{P\text{-}tracks} = \{a,\ b{\cdot}x\}$	$\mathtt{P\text{-}tracks} = \{a,\ b{\cdot}x\}$

Figure 5.9: Examples of *tracks*, *P-tracks* and *T-tracks*.

Example 5.37
In figure 5.9 we see that $S_1 \not\approx_{\mathcal{Q}} S_2$, because $\mathcal{O}_{S_1}(a) = \{x, \delta\} \neq \mathcal{O}_{S_2}(a) = \{x\}$. In our tracks-representation this is reflected by the fact that $a \in \mathit{P\text{-}tracks}(S_1)$, but not $a \in \mathit{P\text{-}tracks}(S_2)$.
$S_3 \approx_{\mathcal{Q}} S_4$, and their tracks are therefore the same.

□

All track-sets of the specifications in example 5.37 are of finite size. So at least for some specifications the tracks-model is finite. This is therefore an improvement over the model as defined in section 5.4, because that one is always infinite. Generally, for specifications with finite behaviour and a finite input alphabet, *P-tracks* and *T-tracks* are also finite, as will be shown further on in this section (remark 5.48).

Relating output deadlocks to the specification

We now investigate how the output deadlocks of a queue context Q_S can be related to the specification S, especially to 'output deadlocks' of S: S **after** σ **refuses** L_U . The implication in one direction holds: output deadlocking traces of a specification S are also output deadlocking traces of its queue context Q_S (proposition 5.38); the other direction is more complicated (proposition 5.41.1).

Proposition 5.38
Let $S \in \mathcal{LTS}$, $\sigma \in L^*$, then S **after** σ **refuses** L_U implies $\delta_S(\sigma)$.

□

In case of an output deadlock $\delta_S(\sigma)$ caused by S **after** σ **refuses** L_U the trace σ is completely consumed by S, so that the input queue is empty. Such a deadlock can sometimes be resolved by putting extra inputs in the input queue: the deadlock

may be, but need not be temporary. Another way to put a queue context in output deadlock is when consumption by S of actions from the input queue is blocked. Such a situation occurs when S cannot consume the first action of the input queue, while S cannot produce any outputs either. The environment cannot resolve this blocking by adding extra inputs: this output deadlock is always permanent. A trace $\sigma{\cdot}a$ is such a permanent, blocking deadlock if S **after** σ **refuses** $\{a\}\cup L_U$:

Proposition 5.39
Let $S \in \mathcal{LTS}$, $\sigma \in L^*$, $a \in L_I$, then

$$S \textbf{ after } \sigma \textbf{ refuses } \{a\}\cup L_U \quad \text{implies} \quad \sigma{\cdot}a \in \delta\text{-}perm(S)$$

□

All deadlock traces caused by proposition 5.38 are collected in δ-*empty*; all those caused by proposition 5.39 are collected in δ-*block*. Note that the closures according to $|@|$ and $@$ are taken into account. *E-tracks* and *B-tracks* are the minimal elements of δ-*empty* and δ-*block* with respect to $|@|$ and $@$ respectively. Note that the alternative definitions of 5.40.3 and 5.40.4 follow directly from proposition A.1. Using these definitions proposition 5.41.1 expresses the relation between a specification S and the output deadlocking traces of its queue context Q_S: it states that all deadlocks are either caused by an empty input queue while S cannot output anything, or by blocking of the input queue. This relation is more complex than the relation between the traces of S and Q_S (cf. corollary 5.23). Comparing with δ-*temp* and δ-*perm* (proposition 5.29) the distinction between δ-*empty* and δ-*block* is not as profitable: δ-*empty* and δ-*block* do not characterize $\approx_{\mathcal{Q}}$.

Definition 5.40

1. $\delta\text{-}empty(S) \;=_{def}$
   $\{\ \sigma \in L^* \mid \exists \sigma' \in traces(S) : \sigma' \,|@|\, \sigma \ \text{ and } \ S \textbf{ after } \sigma' \textbf{ refuses } L_U\ \}$
2. $\delta\text{-}block(S) \;=_{def}$
   $\{\ \sigma \in L^* \mid \exists \sigma' \in traces(S), a \in L_I : \sigma'{\cdot}a \,@\, \sigma \ \text{ and } \ S \textbf{ after } \sigma' \textbf{ refuses } \{a\}\cup L_U\ \}$
3. $\begin{aligned} E\text{-}tracks(S) \;&=_{def}\; min_{|@|}(\delta\text{-}empty(S)) \\ &=\; min_{|@|}(\{\sigma \in L^* \mid S \textbf{ after } \sigma \textbf{ refuses } L_U\ \}) \end{aligned}$
4. $\begin{aligned} B\text{-}tracks(S) \;&=_{def}\; min_{@}(\delta\text{-}block(S)) \\ &=\; min_{@}(\{\sigma{\cdot}a \in L^* \times L_I \mid S \textbf{ after } \sigma \textbf{ refuses } \{a\}\cup L_U\ \}) \end{aligned}$

□

Proposition 5.41

1. $\delta\text{-}traces(S) \;=\; \delta\text{-}empty(S) \cup \delta\text{-}block(S)$
2. Not for all S: $\delta\text{-}empty(S) \cap \delta\text{-}block(S) \;=\; \emptyset$
3. $S_1 \approx_{\mathcal{Q}} S_2$ does not imply $\delta\text{-}empty(S_1) = \delta\text{-}empty(S_2)$,
   nor $\delta\text{-}block(S_1) = \delta\text{-}block(S_2)$

□

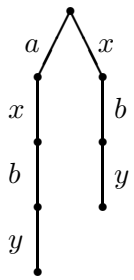


Figure 5.10: Relating output deadlocks to S.

Example 5.42
Consider figure 5.10. Suppose we want to know if the queue context can get in output deadlock after $a{\cdot}x{\cdot}b$. There is no output deadlock with 'empty input queue' which can cause an output deadlock: the only trace of S which is in $|@|$-relation with $a{\cdot}x{\cdot}b$ is $a{\cdot}x{\cdot}b$ itself, and S **after** $a{\cdot}x{\cdot}b$ **must** L_U . However, there is a blocking deadlock which can cause an output deadlock: S **after** x **refuses** $\{a\}{\cup}L_U$, and $x{\cdot}a \,@\, a{\cdot}x{\cdot}b$. Therefore $\delta_S(a{\cdot}x{\cdot}b)$, and this deadlock is permanent.
□

Example 5.43
In figure 5.10, $ax \in \delta\text{-}empty(S)$ and $ax \in \delta\text{-}block(S)$, hence $\delta\text{-}empty(S){\cap}\delta\text{-}block(S){\neq}\emptyset$.

In figure 5.9, $S_3 \approx_{\mathcal{Q}} S_4$, but $a \in \delta\text{-}empty(S_3)$ and $a \notin \delta\text{-}empty(S_4)$.
Moreover, $a \notin \delta\text{-}block(S_3)$ and $a \in \delta\text{-}block(S_4)$.
□

Relating *T-tracks* and *P-tracks* to *E-tracks* and *B-tracks*

According to theorem 5.36, *tracks*, *T-tracks*, and *P-tracks* fully characterize $\approx_{\mathcal{Q}}$. However, *T-tracks* and *P-tracks* cannot easily be obtained from a specification S: they are defined in terms of the queue context Q_S. *E-tracks* and *B-tracks* are defined in terms of the specification, however, they do not characterize $\approx_{\mathcal{Q}}$. This leads to the desirability to relate *T-tracks* and *P-tracks* to *E-tracks* and *B-tracks*, in particular to obtain *T-tracks* and *P-tracks* from *E-tracks* and *B-tracks*.

As already noted, 'blocking' deadlocks are always permanent, but for 'empty-input-queue' deadlocks it cannot trivially be decided whether they are elements of $\delta\text{-}perm$ or of $\delta\text{-}temp$. A *T-track* is always a trace of S, but a *P-track* need not be a trace of S. One might suspect that *P-tracks* are very closely related to traces of S, e.g.

$$\sigma \in \textit{P-tracks}(S) \quad \text{implies} \quad \sigma \in \textit{traces}(S) \;\text{ or }\; (\,\exists \sigma', a : \sigma = \sigma'{\cdot}a \text{ and } \sigma' \in \textit{traces}(S)\,)$$

but example 5.45 shows that this is not correct. The relation between *P-tracks* and traces of S is expressed by proposition 5.44.5.

Proposition 5.44

1. $\delta\text{-}block(S) \subseteq \delta\text{-}perm(S)$
2. $\delta\text{-}temp(S) \subseteq \delta\text{-}empty(S)$
3. $T\text{-}tracks(S) = E\text{-}tracks(S) \cap \delta\text{-}temp(S)$ (and hence $T\text{-}tracks(S) \subseteq E\text{-}tracks(S)$)
4. $\sigma \in T\text{-}tracks(S)$ implies S **after** σ **refuses** L_U (and hence $\sigma \in traces(S)$)
5. $\sigma \in P\text{-}tracks(S)$ implies
   $\sigma \in \delta\text{-}empty(S)$
   or ($\exists \sigma' \in traces(S), a \in L_I : \sigma' \cdot a = \sigma$ and S **after** σ' **refuses** $\{a\} \cup L_U$)
6. $\sigma \in \delta\text{-}perm(S)$ iff $\delta_S(\sigma)$ and $\forall a \in L_I : \sigma \cdot a \in \delta\text{-}perm(S)$

□

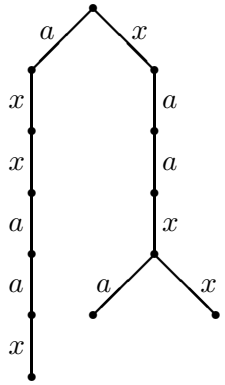


Figure 5.11: Example of computing the tracks of a specification.

Example 5.45
Let S be the specification of figure 5.11, with $L_I = \{a\}, L_U = \{x\}$.
Now we have $a \cdot x \cdot a \cdot x \in P\text{-}tracks(S)$, but $a \cdot x \cdot a \cdot x \notin traces(S)$:

$\delta_S(a{\cdot}x{\cdot}a{\cdot}x)$, since S **after** $a{\cdot}x{\cdot}x{\cdot}a$ **refuses** L_U and $a{\cdot}x{\cdot}x{\cdot}a \,|@|\, a{\cdot}x{\cdot}a{\cdot}x$ (propositions 5.38 and 5.33.6). Moreover, $a \cdot x \cdot a \cdot x \cdot a \in \delta\text{-}perm(S)$, since $x \cdot a \cdot a \cdot x \cdot a \in \delta\text{-}perm(S)$ and $x{\cdot}a{\cdot}a{\cdot}x{\cdot}a \,@\, a{\cdot}x{\cdot}a{\cdot}x{\cdot}a$ (proposition 5.44.6), and $a{\cdot}x{\cdot}a{\cdot}x$ is @-minimal in $\delta\text{-}perm(S)$, because $a{\cdot}x{\cdot}x$ and $a{\cdot}x{\cdot}x{\cdot}a$, the only $\delta\text{-}traces(S)$ @-smaller than $a{\cdot}x{\cdot}a{\cdot}x$, are both elements of $\delta\text{-}temp(S)$ (they alway produce output x after some a's).

□

Example 5.46
$\delta\text{-}block(S) \subseteq \delta\text{-}perm(S)$ (proposition 5.44.1), but $B\text{-}tracks(S) \not\subseteq P\text{-}tracks(S)$: in figure 5.9, $a{\cdot}b \in B\text{-}tracks(S_1)$, but $a{\cdot}b \notin P\text{-}tracks(S_1)$, since $a \in P\text{-}tracks(S_1)$ and $a \,@\, a{\cdot}b$.

□

From proposition 5.44.3 it follows that $T\text{-}tracks$ is obtained from $E\text{-}tracks$ by removing traces that are not in $\delta\text{-}temp$, i.e. traces in $\delta\text{-}perm$, while $\delta\text{-}perm$ is obtained from

δ-*block* by adding traces in δ-*empty* that are also in δ-*perm*:

$$\delta\text{-}perm(S) \;=\; \delta\text{-}block(S) \cup (\, \delta\text{-}empty(S) \cap \delta\text{-}perm(S)\,)$$

from which *P-tracks* is obtained as

$$\begin{array}{rcl} P\text{-}tracks(S) & = & min_{@}(\delta\text{-}perm(S)) \\ & = & min_{@}(\delta\text{-}block(S) \cup (\, \delta\text{-}empty(S) \cap \delta\text{-}perm(S)\,)\,) \\ & = & min_{@}(B\text{-}tracks(S) \cup (\, \delta\text{-}empty(S) \cap \delta\text{-}perm(S)\,)\,) \end{array}$$

This suggests a procedure for obtaining *T-tracks* and *P-tracks* for specifications with finite set of traces: starting with δ-*empty* and *B-tracks*, traces that belong in δ-*perm* are moved from δ-*empty* to *B-tracks*. If no more traces can be moved, then δ-*empty* is equal to δ-*temp*, and *B-tracks* is equal to *P-tracks*. Proposition 5.44.6 gives a criterion for traces to be moved:

1. Let $\begin{array}{rcl} X & = & \delta\text{-}empty(S) \\ Y & = & B\text{-}tracks(S) \end{array}$

   It follows that $\begin{array}{lll} X \supseteq \delta\text{-}temp(S) & (*\;\; 5.44.2 \;\; *) \\ \overline{Y} \subseteq \delta\text{-}perm(S) & (*\;\; 5.44.1 \;\; *) \\ X \cup \overline{Y} = \delta\text{-}traces(S) & (*\;\; 5.41.1 \;\; *) \end{array}$

2. Remove from X all traces that are in $\delta\text{-}block(S) = \overline{Y}$, then the assertions of 1. are not violated;

3. If $\sigma \in X$ such that $\forall a \in L_I : \sigma{\cdot}a \in \overline{Y}$, then $(*\;\; 5.44.6 \;\; *)$ $\sigma \in \delta\text{-}perm(S)$:

   $\begin{array}{rcl} X & := & X \backslash \{\sigma\} \\ Y & := & min_{@}(Y \cup \{\sigma\}) \end{array}$

   The assertions are still valid.

4. If there are no more $\sigma \in X$ that can be moved to $\overline{Y}$ $(*$ finiteness $X \;\; *)$, then $(*\;\; 5.44.6 \;\; *)$ $X \cap \delta\text{-}perm(S) = \emptyset$. Together with $X \supseteq \delta\text{-}temp(S)$ it follows that $X = \delta\text{-}temp(S)$, and $T\text{-}tracks(S) = min_{|@|\,(X)}$.

   Moreover, it follows that $\overline{Y} = \delta\text{-}perm(S)$, $P\text{-}tracks(S) = min_{@}(Y)$.

The procedure is illustrated in example 5.47.

Example 5.47
Let S be given by figure 5.6, with $L_I = \{a, b\}$ and $L_U = \{x, y\}$.

1. We start with computing $X = \delta\text{-}empty(S)$ and $Y = B\text{-}tracks(S)$:

$$\begin{array}{rcl} X & = & \delta\text{-}empty(S) \\ & = & \{\sigma \in L^* \mid \exists \sigma' \in traces(S) : \sigma' \,|@|\, \sigma \text{ and } S \textbf{ after } \sigma' \textbf{ refuses } L_U \,\} \\ & = & \{\; \epsilon,\; a{\cdot}x,\; a{\cdot}x{\cdot}b{\cdot}y,\; a{\cdot}b{\cdot}x{\cdot}y \;\} \\ Y & = & B\text{-}tracks(S) \\ & = & min_{@}(\{\sigma{\cdot}a \in L^* \times L_I \mid \; S \textbf{ after } \sigma \textbf{ refuses } \{a\} \cup L_U \,\}) \\ & = & \{\; b,\; a{\cdot}x{\cdot}a,\; a{\cdot}x{\cdot}b{\cdot}y{\cdot}a,\; a{\cdot}x{\cdot}b{\cdot}y{\cdot}b \;\} \end{array}$$

2. There are no elements in X that are also in $\overline{Y}$.
3. Now we move elements from X to Y:

   $a{\cdot}b{\cdot}x{\cdot}y \in X$ and $a{\cdot}b{\cdot}x{\cdot}y{\cdot}a \in \overline{Y}$ and $a{\cdot}b{\cdot}x{\cdot}y{\cdot}b \in \overline{Y}$, so we can move $a{\cdot}b{\cdot}x{\cdot}y$ from X to Y:

   $$\begin{array}{lcl} X' & = & \{\ \epsilon,\ a{\cdot}x,\ a{\cdot}x{\cdot}b{\cdot}y\ \} \\ Y' & = & \{\ b,\ a{\cdot}x{\cdot}a,\ a{\cdot}x{\cdot}b{\cdot}y{\cdot}a,\ a{\cdot}x{\cdot}b{\cdot}y{\cdot}b,\ a{\cdot}b{\cdot}x{\cdot}y\ \} \end{array}$$

   With similar reasoning we deduce that $a{\cdot}x{\cdot}b{\cdot}y \in X'$ can be moved to the Y':

   $$\begin{array}{lcl} X'' & = & \{\ \epsilon,\ a{\cdot}x\ \} \\ Y'' & = & \{\ b,\ a{\cdot}x{\cdot}a,\ a{\cdot}x{\cdot}b{\cdot}y{\cdot}a,\ a{\cdot}x{\cdot}b{\cdot}y{\cdot}b,\ a{\cdot}b{\cdot}x{\cdot}y,\ a{\cdot}x{\cdot}b{\cdot}y\ \} \end{array}$$

   With the addition of $a{\cdot}x{\cdot}b{\cdot}y$, Y'' now contains non-minimal elements which can be removed:

   $$Y''' \ = \ \{\ b,\ a{\cdot}x{\cdot}a,\ a{\cdot}x{\cdot}b{\cdot}y\ \}$$

4. No other elements can be moved from X'' to Y''', thus

   $$\begin{array}{lcllcl} \textit{T-tracks}(S) & = & \textit{min}_{|@|}(X'') & = & \{\ \epsilon,\ a{\cdot}x\ \} \\ \textit{P-tracks} & = & Y''' & = & \{\ b,\ a{\cdot}x{\cdot}a,\ a{\cdot}x{\cdot}b{\cdot}y\ \} \end{array}$$

□

Remark 5.48
Proposition 5.44 shows that for specifications with finite behaviour ($\textit{traces}(S)$ is finite) and with a finite input alphabet both *T-tracks* and *P-tracks* are finite. For *T-tracks* this follows easily from proposition 5.44.4: *T-tracks* $\subseteq$ $\textit{traces}(S)$ (using that S **after** σ **refuses** L_U implies $\sigma \in \textit{traces}(S)$). For *P-tracks* it follows from proposition 5.44.5:

$$\textit{P-tracks}(S) \subseteq \{\sigma \in L^* \mid \exists \sigma' \in \textit{traces}(S) : \sigma' \,|@|\, \sigma\} \cup \{\sigma{\cdot}a \mid \sigma \in \textit{traces}(S), a \in L_I\}$$

which are both finite sets.

□

5.7 Queue Implementation Relations

After studying queue equivalence in the previous sections we now consider non-equivalence relations over $\mathcal{LTS}$ in queue contexts. The main focus is on relations that can be used as implementation relations.

The first candidates for implementation relations are the queue context relations (definition 5.3) with respect to the implementation relations based on synchronous communication $\leq_{te}$, $\leq_{tr}$, and **conf** (definitions 1.15.1, 3.8, and 3.12).

Definition 5.49

1. $I \ \leq_{te}^{Q} \ S \ =_{def} \ Q_I \ \leq_{te} \ Q_S$

2. $I \leq_{tr}^{Q} S \;=_{def}\; Q_I \leq_{tr} Q_S$
3. $I \,\mathbf{conf}_Q\, S \;=_{def}\; Q_I \,\mathbf{conf}\, Q_S$

□

The next possible candidates are inspired by the characterization of queue equivalence in theorem 5.14. Apart from $\approx_{\mathcal{Q}}$ itself as an implementation relation we can consider relations where the $=$-symbol is replaced by $\subseteq$ or $\supseteq$. We elaborate on $\subseteq$, since it matches with our intuition of an implementation relation: if the implementation produces an output x then the specification must be able to produce the same output x, and the implementation may only reach an output deadlock $\delta_I(\sigma)$, i.e. produce no output if the specification does so. The relation thus defined is called *queue preorder* $\leq_{\mathcal{O}}$. We also consider the relations that arise from restricting to outputs and to deadlocks.

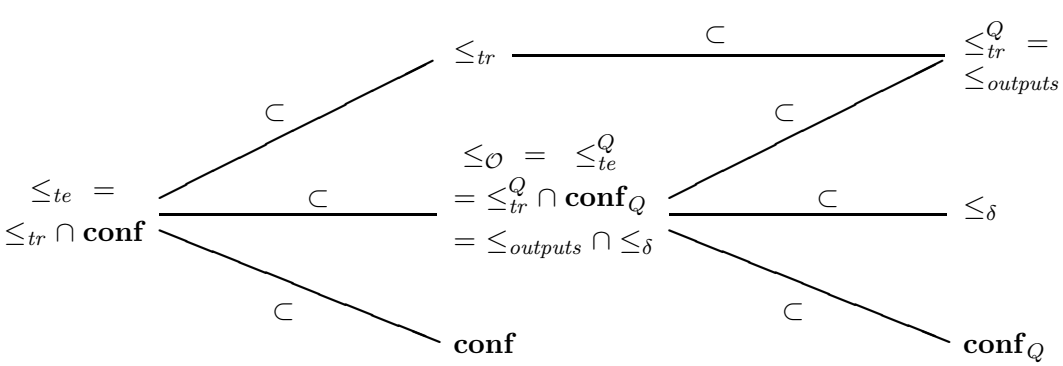
Definition 5.50

1. $I \leq_{\mathcal{O}} S \;=_{def}\; \forall \sigma \in L^* : \mathcal{O}_I(\sigma) \subseteq \mathcal{O}_S(\sigma)$
2. $I \leq_{outputs} S \;=_{def}\; \forall \sigma \in L^* : outputs_I(\sigma) \subseteq outputs_S(\sigma)$
3. $I \leq_{\delta} S \;=_{def}\; \forall \sigma \in L^* : \delta_I(\sigma)$ implies $\delta_S(\sigma)$

□

The following proposition relates the asynchronous and synchronous implementation relations. Note that the inclusions are strict.

Proposition 5.51

1. 

2. For implementations with finite behaviour: $\leq_{\delta} = \leq_{\mathcal{O}}$

□

Example 5.52
First consider figure 5.5:

- $S_1 \approx_{\mathcal{Q}} S_2$, so $\leq_{\mathcal{O}}$, $\leq_{tr}^{Q}$, and $\mathbf{conf}_Q$ hold in both directions, but none of the synchronous implementation relations holds. This shows that the inclusion $\leq_{te} \subset \leq_{\mathcal{O}}$ is strict.
- Since $T_1 \approx_{\mathcal{Q}} T_2$, also here $\leq_{\mathcal{O}}$, $\leq_{tr}^{Q}$, and $\mathbf{conf}_Q$ hold in both directions. Moreover, $T_1 \leq_{te} T_2$, but $T_2 \not\leq_{te} T_1$, hence again: $\leq_{te} \subset \leq_{\mathcal{O}}$.

- $\mathcal{O}_{U_1}(a) = \{x\} \subseteq \{x, \delta\} = \mathcal{O}_{U_2}(a)$, and analogous for b, while for other traces of U_1: $\mathcal{O}_{U_1}(\sigma) = \mathcal{O}_{U_2}(\sigma)$, so $U_1 \leq_{\mathcal{O}} U_2$ and $U_2 \not\leq_{\mathcal{O}} U_1$. Also $U_1 \leq_{te} U_2$ and $U_2 \not\leq_{te} U_1$.

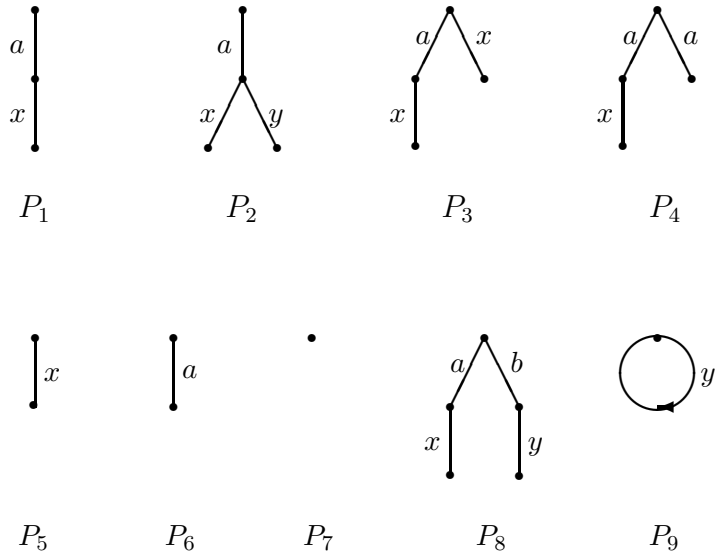


Figure 5.12: Examples of implementations and specifications.

In figure 5.12:

- $\leq_{\mathcal{O}}$ is not related to $\leq_{tr}$, nor to **conf**: $P_6 \leq_{\mathcal{O}} P_7$, but $P_6 \not\leq_{tr} P_7$, and $P_4 \leq_{tr} P_1$, but $P_4 \not\leq_{\mathcal{O}} P_1$; P_2 **conf** P_1, but $P_2 \not\leq_{\mathcal{O}} P_1$, and $P_1 \leq_{\mathcal{O}} P_2$, but P_1 $\not$**conf** P_2.

  Since $\leq_{\mathcal{O}}$ is not related to **conf**, the test cases derived according to the algorithms of chapter 4 are not sound with respect to asynchronous testing: the test case $a; y;$ **stop** of P_2 rejects P_1 as a correct implementation.
- The same example of P_1 and P_2 applies to show that **conf** is not related to $\mathbf{conf}_Q$.
- $\leq_{\mathcal{O}}$ is strictly contained in $\mathbf{conf}_Q$ and $\leq_{tr}^{Q}$: P_3 $\mathbf{conf}_Q P_1$, while $P_3 \not\leq_{\mathcal{O}} P_1$, and $P_4 \leq_{tr}^{Q} P_1$, while $P_4 \not\leq_{\mathcal{O}} P_1$.
- In implementations $\leq_{\mathcal{O}}$ does not allow extra outputs: $P_2 \not\leq_{\mathcal{O}} P_1$, nor absence of outputs when outputs are specified: $P_4 \not\leq_{\mathcal{O}} P_1$, nor spontaneous outputs when no outputs are allowed: $P_3 \not\leq_{\mathcal{O}} P_1$. Combining these requirements we see that the only correct implementation of P_1 according to $\leq_{\mathcal{O}}$ is P_1 itself.
- As opposed to **conf** and $\mathbf{conf}_Q$, $\leq_{tr}$ and $\leq_{tr}^{Q}$ are related: $\leq_{tr} \subset \leq_{tr}^{Q}$. They are not equal: $P_6 \leq_{tr}^{Q} P_7$, but $P_6 \not\leq_{tr} P_7$.
- Adding a branch $a{\cdot}x$ to the output action x does not change the process: $P_3 \approx_{\mathcal{Q}} P_5$, but adding the same branch to the input action a changes the process: $P_4 \not\approx_{\mathcal{Q}} P_6$.
- P_9 spontaneously generates outputs; it has no deadlocks. Hence $P_9 \leq_{\delta} R$ for any R, in particular $P_9 \leq_{\delta} P_5$ and $P_9 \leq_{\delta} P_7$, whereas $P_9 \not\leq_{\mathcal{O}} P_5$ and $P_9 \not\leq_{\mathcal{O}} P_7$. Like P_9, all specifications that can distinguish between $\leq_{\delta}$ and $\leq_{\mathcal{O}}$ have infinite behaviour.

□

The relation $\leq_{\mathcal{O}}$ seems to be an interesting implementation relation in queue contexts: in example 5.52 we saw that $\mathbf{conf}_Q$ sometimes allows spontaneous outputs, $\leq_{tr}^{Q}$ ($=\leq_{outputs}$) does not preserve deadlock behaviour, and $\leq_{\delta}$ only requires that an output occurs, no matter what output. The implementation relation $\leq_{\mathcal{O}}$ matches with our intuition of what constitutes a correct implementation in a queue context. Moreover, it is implied by and follows naturally from the important synchronous relation $\leq_{te}$.

However, analogous to the synchronous relation $\leq_{te}$, $\leq_{\mathcal{O}}$ has a drawback in using it as a basis for conformance testing of implementations with respect to specifications: it is defined using a quantification $\forall \sigma \in L^*$. For the purpose of testing this poses the problem of having to verify $\mathcal{O}_I(\sigma) \subseteq \mathcal{O}_S(\sigma)$ for all possible traces in L^*. A reduction of the number of traces for which to test is desirable. For synchronous conformance testing the coarser relation **conf** was defined to cope with this problem. The relation **conf** is obtained from $\leq_{te}$ by reducing the quantification to the traces of the specification S (definition 3.12).

An analogous approach can be taken for $\leq_{\mathcal{O}}$: we consider relations $\leq_{\mathcal{F}}$ with $\mathcal{F}$ a subset of L^*, which would typically depend on S:

$$I \leq_{\mathcal{F}} S \quad =_{def} \quad \forall \sigma \in \mathcal{F} : \mathcal{O}_I(\sigma) \subseteq \mathcal{O}_S(\sigma) \tag{5.2}$$

The first obvious choice for $\mathcal{F}$ is $traces(S)$, defining the relation $\leq_{tr(S)}$:

$$I \leq_{tr(S)} S \quad =_{def} \quad \forall \sigma \in traces(S) : \mathcal{O}_I(\sigma) \subseteq \mathcal{O}_S(\sigma) \tag{5.3}$$

The relation $\leq_{tr(S)}$ has an undesirable feature; equivalent specifications do not have the same conforming implementations, i.e. $\leq_{tr(S)}$ does not satisfy:

$$S_1 \approx_{\mathcal{Q}} S_2 \text{ implies } (I \leq_{tr(S)} S_1 \text{ iff } I \leq_{tr(S)} S_2) \tag{5.4}$$

Consider figure 5.13. $S_1 \approx_{\mathcal{Q}} S_2$, but $I_2 \leq_{tr(S)} S_1$ and $I_2 \not\leq_{tr(S)} S_2$: $\mathcal{O}_{S_1}(a{\cdot}b) = \mathcal{O}_{S_2}(a{\cdot}b) = \{x\}$; $\mathcal{O}_{I_2}(a{\cdot}b) = \{x, y\}$. However, for $I_2 \leq_{tr(S)} S_1$, $\mathcal{O}_{I_2}(a{\cdot}b) \subseteq \mathcal{O}_{S_1}(a{\cdot}b)$ is not required, since $a{\cdot}b \notin traces(S_1)$.

A next possible choice is $\mathcal{F} = traces(Q_S)$, defining the relation $\leq_{tr(Q_S)}$. Since $S_1 \approx_{\mathcal{Q}} S_2$ implies $traces(Q_{S_1}) = traces(Q_{S_2})$ (corollary 5.15) it is easy to check that the analogue of (5.4) is satisfied. However, as we will see in proposition 5.55 this new relation is equal to $\leq_{\mathcal{O}}$.

Also $tracks(S)$ can be considered as the basis for an implementation relation. From theorem 5.26 we know that two equivalent specifications have the same tracks, thus requirement (5.4) is satisfied for any set $\mathcal{F}$ that only depends on $tracks(S)$.

There are different possibilities for considering the tracks as the basis for an implementation relation. The first choice is $\mathcal{F} = tracks(S)$ in (5.2), but then P_6 would conform to P_1 (figure 5.12), which looks counterintuitive. Conformance is caused by the fact that the tracks of P_1 are ϵ and $a{\cdot}x$, which both have $\mathcal{O}_{P_1}(\sigma) = \{\delta\}$. There is no requirement that after a the output x shall occur. This suggests that we should not test

for tracks of the specification, but for traces for which an output can be expected, i.e. traces σ such that $\sigma{\cdot}x$ is a track. Note that tracks are always equal to ϵ or end with an output action (proposition 5.25.4). The relation thus defined is called **asco**. There are no requirements on spontaneous outputs from the implementation at points where no output was expected according to the tracks.

The relation **aconf** is closely related to **asco**, the difference being that it does check for spontaneous outputs for all prefixes of $tracks(S)$, also those where no output is expected.

Definition 5.53

1. I **asco** $S \;=_{def}\; \forall\sigma \in L^*, \forall x \in L_U :$ if $\sigma{\cdot}x \in tracks(S)$ then $\mathcal{O}_I(\sigma) \subseteq \mathcal{O}_S(\sigma)$
2. I **aconf** $S \;=_{def}\; \forall\sigma \in tracks(S), \forall\sigma' \in L^* :$ if $\sigma' \preceq \sigma$ then $\mathcal{O}_I(\sigma') \subseteq \mathcal{O}_S(\sigma')$

□

Proposition 5.54
asco and **aconf** are reflexive, but not transitive.

□

The relations $\leq_{\mathcal{O}}$, $\leq_{tr(Q_S)}$, $\leq_{tr(S)}$, **aconf**, **asco** are related in the following proposition.

Proposition 5.55

1. $\leq_{\mathcal{O}} \;=\; \leq_{L^*} \;=\; \leq_{tr(Q_S)} \;\subset\; \leq_{tr(S)} \;\subset\;$ **aconf** $\;\subset\;$ **asco**
2. The relations **asco** and **aconf** do not contain, nor are contained in $\leq_{tr}^{Q}$ ($\leq_{outputs}$), $\mathbf{conf}_Q$, or $\leq_{\delta}$.

□

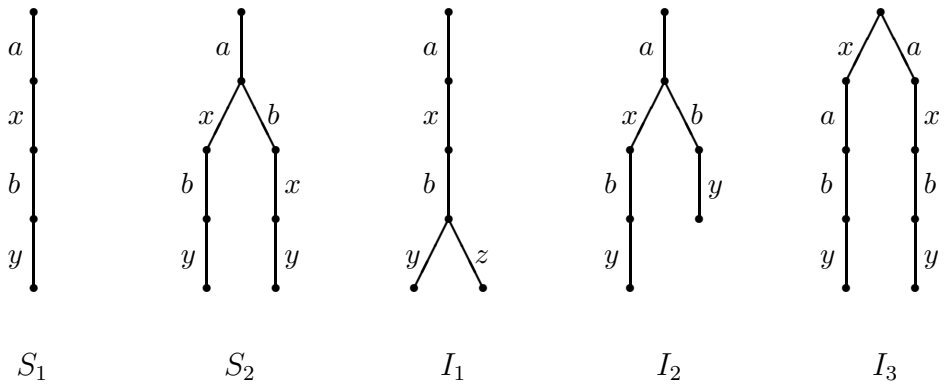


Figure 5.13: Implementations for **asco** and **aconf**.

Example 5.56
In figure 5.12: For **asco**, P_1 only specifies that after a an output x must occur: P_3, P_5, P_8 **asco** P_1.
For **aconf**, P_1 also specifies that after ϵ and $a{\cdot}x$ no output may occur: P_3, P_5 **aconf** P_1, and P_8 **aconf** P_1.

In figure 5.13: I_1 **aco$\not$nf** S_1, because $\mathcal{O}_{I_1}(a{\cdot}x{\cdot}b) = \{y, z\} \not\subseteq \mathcal{O}_{S_1}(a{\cdot}x{\cdot}b) = \{y\}$. I_2 **aconf** S_1, although $I_2 \not\leq_{\mathcal{O}} S_1$. I_3 **aco$\not$nf** S_1 because $\mathcal{O}_{I_3}(\epsilon) = \{x\} \not\subseteq \mathcal{O}_{S_1}(\epsilon) = \{\delta\}$.

We also have I_1 **as$\not$co** S_1, and I_2 **asco** S_1, but I_3 **asco** S_1: we do not check $\mathcal{O}_{I_3}(\epsilon)$, because we do not expect any output until we provide the input a. **aconf** does test for such unexpected outputs.

□

Other implementation relations can be defined by taking other sets $\mathcal{F}$ that depend on $tracks(S)$, e.g. defining $\mathcal{F}$ based on the relations @ or $|@|$. As an example a family of implementation relations $\mathbf{aconf}^n$, $n \in \mathbf{N}$, is introduced. These implementation relations gradually range from **aconf** (for $n = 0$) to $\leq_{\mathcal{O}}$ (in the limiting case). In **aconf** we require $\mathcal{O}_I(\sigma) \subseteq \mathcal{O}_S(\sigma)$ for all prefixes of $tracks(S)$; in $\leq_{\mathcal{O}}$ this is required for all traces in L^*. According to proposition 5.55.1 this corresponds to all traces in $traces(Q_S)$, i.e. all traces that are in @-relation to $tracks(S)$ (proposition 5.25.2). These traces can be obtained from tracks by shifting input actions to the front and adding input actions at the end. The family of relations $\mathbf{aconf}^n$ is defined by limiting the number of shifts and additions of actions with respect to the tracks. This number of shifts and additions is defined by the relation $@^n$: $\sigma_1 @^n \sigma_2$ iff σ_2 can be obtained from σ_1 by exactly n shifts or additions.

Definition 5.57
For n a natural number,

1. $@^n \subseteq L^* \times L^*$ is defined as the smallest subset of @ satisfying
   - if $\sigma_1, \sigma_2 \in L_I^*$ then $\sigma_1 @^n \sigma_2 \;\; =_{def} \;\; |\sigma_2 \backslash\backslash \sigma_1| = n$
   - if $\sigma_1 = \rho_1 {\cdot} x {\cdot} \sigma_1', \sigma_2 = \rho_2 {\cdot} x {\cdot} \sigma_2'$, with $\rho_1, \rho_2 \in L_I^*, x \in L_U, \sigma_1', \sigma_2' \in L^*$, then $\sigma_1 @^n \sigma_2 \;\; =_{def} \;\; \exists m \leq n : \rho_1 @^m \rho_2$ and $\sigma_1' \;@^{n-m} (\rho_2 \backslash \rho_1) {\cdot} \sigma_2'$
2. The *tracks of order n* of a specification S are defined by
   $tracks^n(S) \;\; =_{def} \;\; \{\sigma \in L^* \mid \exists \sigma' \in tracks(S) : \sigma' \;@^n \sigma\}$
3. The n^{th} order of **aconf** is defined by
   I $\mathbf{aconf}^n S \;=_{def}\; \forall \sigma \in \bigcup_{i=0}^{n} \; tracks^i(S), \forall \sigma' \in L^*$: if $\sigma' \preceq \sigma$ then $\mathcal{O}_I(\sigma') \subseteq \mathcal{O}_S(\sigma')$

□

Proposition 5.58

1. $\sigma_1 @^0 \sigma_2$ iff $\sigma_1 = \sigma_2$
2. $\bigcup_{n=0}^{\infty} \; @^n = @$
3. $tracks^0(S) = tracks(S)$
4. $tracks^n(S) \subseteq traces(Q_S)$
5. $\bigcup_{n=0}^{\infty} \; tracks^n(S) = traces(Q_S)$
6. $\mathbf{aconf}^0 = \mathbf{aconf}$
7. $\bigcap_{n=0}^{\infty} \; \mathbf{aconf}^n = \leq_{\mathcal{O}}$
8. $m \leq n$ implies $\mathbf{aconf}^m \supseteq \mathbf{aconf}^n$

□

Example 5.59
In figure 5.13, I_2 **aconf** S_1 (example 5.56). However, I_2 **aco⁄nf** $^1S_1$:
$tracks^1(S_1) = \{a,\ b,\ a{\cdot}x{\cdot}a,\ a{\cdot}x{\cdot}b,\ a{\cdot}x{\cdot}b{\cdot}y{\cdot}a,\ a{\cdot}x{\cdot}b{\cdot}y{\cdot}b,\ a{\cdot}b{\cdot}x{\cdot}y\}$.
$a{\cdot}b$ is a prefix of $a{\cdot}b{\cdot}x{\cdot}y \in tracks^1(S_1)$, and $\mathcal{O}_{I_2}(a{\cdot}b) = \{x, y\} \not\subseteq \{x\} = \mathcal{O}_{S_1}(a{\cdot}b)$.
We see that $tracks^1(S_1)$ is already more than twice as large as $tracks(S_1)$. The number of traces to be considered for $\mathbf{aconf}^n$ increases rapidly with increasing n.
□

Different implementation relations allow different classes of conforming implementations. Different criteria may be used to choose a suitable implementation relation. If in example 5.59 it is unacceptable that I_2 is accepted as a correct implementation of S_1, then a more discriminating implementation relation than **aconf** must be chosen, e.g. $\mathbf{aconf}^1$. If $\leq_{\mathcal{O}}$ is considered as the ideal implementation relation, which is not testable due to the infinite number of traces, then the choice of a weaker implementation relation, like **asco** or $\mathbf{aconf}^n$ for some n, corresponds to a kind of test case selection (cf. chapter 6).

5.8 Test Derivation

In this section we deal with test derivation for the family of implementation relations $\leq_{\mathcal{F}}$ ($\mathcal{F} \subseteq L^*$, (5.2)). The approach taken is by first considering test cases for sets containing only one trace: $\mathcal{F} = \{\sigma\}$. Combining all test cases for every trace in $\mathcal{F}$ gives a test suite for $\leq_{\mathcal{F}}$.

Our test cases are written in $\mathcal{DLTS}$ (section 3.4.3, definition 3.25.1). Passing a test is expressed by $\mathbf{passes}_{\mathcal{D}}$ (definition 3.25.2), however, a test case does not observe the implementation I but the queue context $[_{\epsilon}{\ll}I{\ll}_{\epsilon}]$. This asynchronous notion of passing a test is denoted by $\mathbf{a\text{-}passes}_{\mathcal{D}}$, and is used in the rest of this section.

Definition 5.60
Let $I \in \mathcal{LTS}$, $t \in \mathcal{DLTS}$, then $\quad I\ \mathbf{a\text{-}passes}_{\mathcal{D}}\ t\ \ =_{def}\ \ [_{\epsilon}{\ll}I{\ll}_{\epsilon}]\ \mathbf{passes}_{\mathcal{D}}\ t$
□

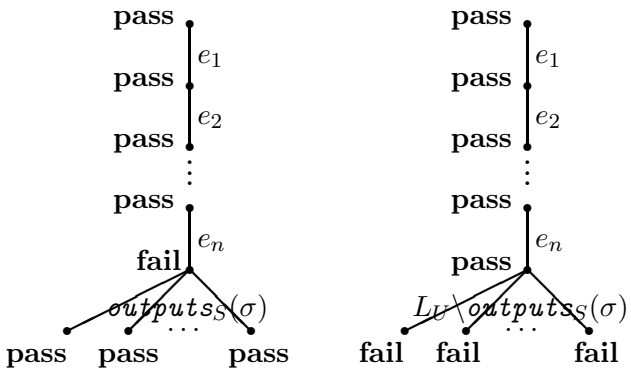
We define the tester $T_S(\sigma)$ and we show that it is a complete tester for $\leq_{\{\sigma\}}$.

Definition 5.61
Let $S \in \mathcal{LTS}$, $\sigma = e_1{\cdot}\ldots{\cdot}e_n \in L^*$, then (figure 5.14)

$T_S(\sigma)\ \ =_{def}$

if not $\delta_S(\sigma)$: $e_1;\ldots;e_n;\Sigma\,\{x;\mathbf{stop} \mid x \in outputs_S(\sigma)\}$,
and, for $\rho \in traces(T_S(\sigma)) : v(T_S(\sigma)\ \mathbf{after}\ \rho) =$ **fail** if $\rho = \sigma$, **pass** if $\rho \neq \sigma$

if $\delta_S(\sigma)$: $e_1;\ldots;e_n;\Sigma\,\{x;\mathbf{stop} \mid x \in L_U \backslash outputs_S(\sigma)\}$,
and, for $\rho \in traces(T_S(\sigma)) : v(T_S(\sigma)\ \mathbf{after}\ \rho) =$ **pass** if $\rho \preceq \sigma$, **fail** if $\rho \not\preceq \sigma$



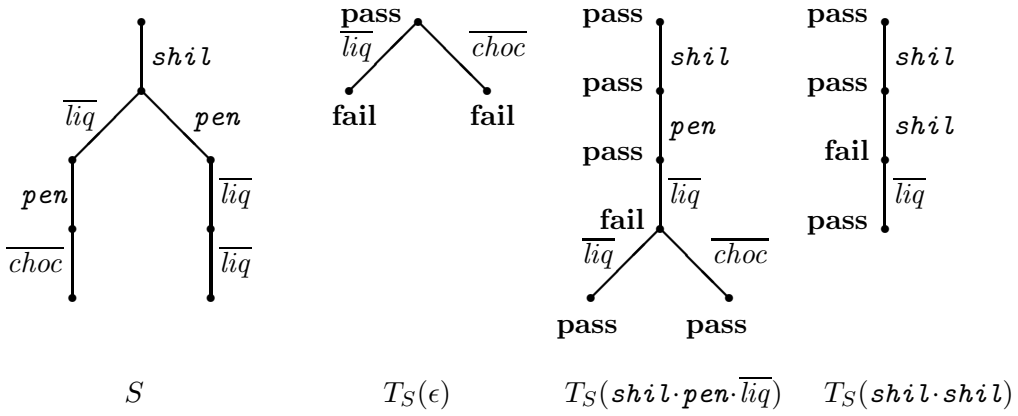
$T_S(\sigma)$, if not $\delta_S(\sigma)$ $T_S(\sigma)$, if $\delta_S(\sigma)$

Figure 5.14: A complete tester for $\leq_{\{\sigma\}}$.

□

Proposition 5.62
$\{T_S(\sigma)\}$ is a complete test suite for S with respect to $\leq_{\{\sigma\}}$.

□



S $T_S(\epsilon)$ $T_S(\mathtt{shil}\cdot\mathtt{pen}\cdot\overline{liq})$ $T_S(\mathtt{shil}\cdot\mathtt{shil})$

Figure 5.15: Examples of test cases.

Example 5.63
In figure 5.15 another candy machine is specified (cf. example 5.2), together with three test cases. These these cases test for $\leq_{\{\epsilon\}}$, $\leq_{\{shil\cdot pen\cdot\overline{liq}\}}$, and $\leq_{\{shil\cdot shil\}}$.

- $T_S(\epsilon)$: The only observation that is made in the initial state is that the machine is in output deadlock: $\delta_S(\epsilon)$; the machine does not give any liquorice or chocolate for free. The test case $T_S(\epsilon)$ tests this: it gives a fail verdict if an implementation gives something for free.
- $T_S(shil\cdot pen\cdot\overline{liq})$: There are two possible observations: the first possibility is that the right branch in the specification is followed, which results in getting another

liquorice. The other possibility is that the penny is still in the tube, while the machine already supplies liquorice: $(shil \cdot \overline{liq} \cdot pen) \,@\, (shil \cdot pen \cdot \overline{liq})$. In this case chocolate is obtained. Either of the two must be supplied, an output deadlock is not allowed. Test case $T_S(shil \cdot pen \cdot \overline{liq})$ tests this.

- $T_S(shil \cdot shil)$: After putting two shillings in the machine the first one will be taken by the machine, while second one remains in the tube. The machine produces liquorice, which is tested by $T_S(shil \cdot shil)$.

□

Using test suites for $\leq_{\{\sigma\}}$ it is straightforward to construct a complete test suite for $\leq_{\mathcal{F}}$ by collecting the test cases $T_S(\sigma)$ for every $\sigma \in \mathcal{F}$:

Theorem 5.64
$\{T_S(\sigma) \mid \sigma \in \mathcal{F}\}$ is a complete test suite for S with respect to $\leq_{\mathcal{F}}$.

□

Theorem 5.64 gives a way of deriving a complete test suite for any implementation relation $\leq_{\mathcal{F}}$. The test cases $T_S(\sigma)$ have some resemblance to the must tests of chapter 3. Like must tests they are not very efficient: during execution of the trace σ no information is extracted from an implementation; only the final branches can detect errors.

Construction of $T_S(\sigma)$ requires computation of the observations $\delta_S(\sigma)$ and $outputs_S(\sigma)$. For simple specifications this is straightforward, but for complex specifications, especially specifications containing a lot of nondeterminism, this may pose problems, as was shown in the previous sections. In the next section we elaborate on an algorithm to compute systematically outputs and deadlocks.

Example 5.65
If we apply the above test derivation algorithm to the specification S of figure 5.16 with implementation relation $\leq_{tr(S)}$, we get the test suite $\Pi_{\leq_{tr(S)}}(S)$ of figure 5.16. In the specification, after *shil* a choice occurs between an input and an output. This is a typical situation where we must be prepared for the peculiarities of asynchronous communication: if the tester tries to input *pen*, the IUT might just have chosen output $\overline{liq}$, and the tester must be aware that this possibility is allowed. We see that the derived test suite handles this case correctly: test case $T_S(shil \cdot pen)$ will not mark the reception of $\overline{liq}$ after inputting *shil* and *pen* as an error.

The tracks of S are $\{\epsilon,\ shil \cdot \overline{liq},\ shil \cdot pen \cdot \overline{choc}\}$, so $\Pi_{\leq_{tr(S)}}(S)$ is also a test suite for **aconf**. A test suite for **asco** consists of only $T_S(shil)$ and $T_S(shil \cdot pen)$. For $\leq_{\mathcal{O}}$ infinitely many test cases have to be added: $T_S(shil \cdot shil \cdot \ldots)$, etc.

□

Translation of test cases to TTCN

The test notation TTCN (section 1.3.3, [ISO91a, part 3]) is based on asynchronous communication. The communication points in TTCN, PCOs, are modelled by unbounded queues. This makes it possible to transform our asynchronous test cases to TTCN. Figure 5.17 gives a scheme for such a transformation. The transformation is given by

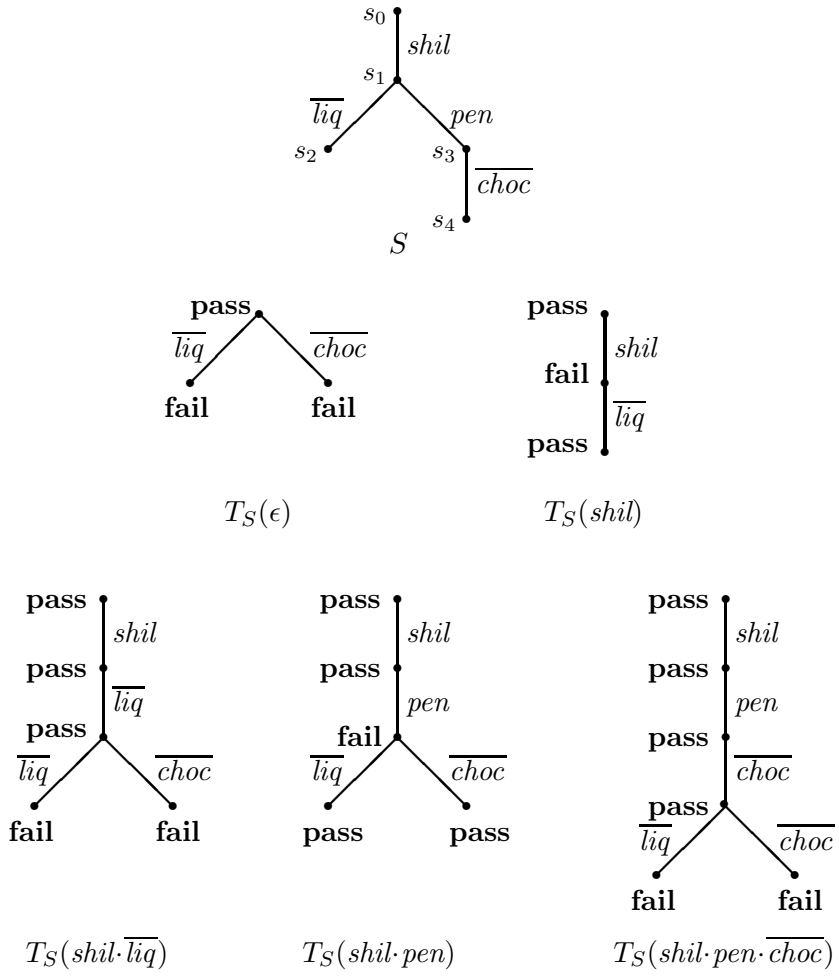
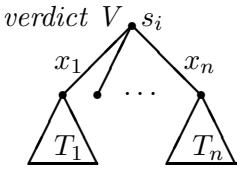


Figure 5.16: S and its test suite $\Pi_{\leq_{tr(S)}}(S)$.

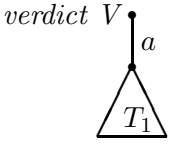
induction on the structure of the tree representing the test case. There is one point of control and observation (PCO) that consists of two queues: one for the output actions of the system (L_U), and one for the input actions (L_I). An output action x is translated to PCO?x: the output action of the system is an input action to the test case, hence the question mark. An input action a is translated to PCO!a. In every state of the TTCN test case in which the tester waits for system outputs a timer must be added to detect output deadlock. In these states we also add a PCO?OTHERWISE to cope with unspecified outputs. If a state is a **pass**-state, we assign verdict pass to the timer expiration and to OTHERWISE, otherwise the verdict is fail. Strictly speaking the TTCN standard stipulates that an OTHERWISE should only lead to a fail verdict. Our use of OTHERWISE should then be taken as a shorthand for the enumeration of all the other actions. Note that the translation of a terminal state only supplies a verdict, which is the verdict of the preceding TTCN line.

- Translation of trees beginning with system outputs



behaviour	verdict
START TIMER_Si	
PCO?x_1	
TTCN(T_1)	
⋮	
PCO?x_n	
TTCN(T_n)	
PCO?OTHERWISE	V
?TIMEOUT TIMER_Si	V

- Translation of trees beginning with system inputs



behaviour	verdict
PCO!a	
TTCN(T_1)	

- Translation of terminal states

verdict V •

behaviour	verdict
	V

Figure 5.17: Translation of a test case to TTCN.

To obtain a correct TTCN test case the translation only makes sense if none of the following three cases occurs:

- nondeterministic choice between actions
- choice between input and output actions
- choice between more than one input action

It is easy to check that these conditions are fulfilled by any $T_S(\sigma)$ (definition 5.61).

Test case $T_S(\boldsymbol{shil}\cdot\overline{liq})$	
behaviour	verdict
PCO!shil	
START Timer1	
PCO?liq	
START Timer2	
PCO?liq	**fail**
PCO?choc	**fail**
PCO?OTHERWISE	**pass**
?TIMEOUT Timer2	**pass**
PCO?OTHERWISE	**pass**
?TIMEOUT Timer1	**pass**

Figure 5.18: TTCN test case $T_S(shil\cdot\overline{liq})$

Example 5.66
If we translate the test cases $T_S(shil\cdot\overline{liq})$ and $T_S(shil\cdot pen)$ of example 5.65 to TTCN, we get the test cases in figures 5.18 and 5.19.
□

Test case $T_S(\boldsymbol{shil}\cdot\boldsymbol{pen})$	
behaviour	verdict
PCO!shil	
PCO!pen	
START Timer	
PCO?liq	**pass**
PCO?choc	**pass**
PCO?OTHERWISE	**fail**
?TIMEOUT Timer	**fail**

Figure 5.19: TTCN test case $T_S(shil\cdot pen)$

5.9 Computation of Outputs and Deadlocks

For the derivation of test cases $outputs_S(\sigma)$ and $\delta_S(\sigma)$ are needed. This section gives a method for their systematic computation. It is based on an extension of proposition 5.22.3: not only every sequence of transitions of a queue context can be replaced by a sequence to a state with empty output queue, but also the $outputs_S(\sigma)$ and $\delta_S(\sigma)$ can be obtained from these states.

We define the *reachability function* μ_S to collect all possible states with empty output queue that can be reached after performing a trace σ. Such a state of a queue context consists of the state of the process in the queue context, together with the contents of the input queue.

Definition 5.67
Let $S \in \mathcal{LTS}$, then the *reachability function* $\mu_S : L^* \rightarrow \mathcal{P}(\mathcal{LTS} \times L_I^*)$ is defined by

$$\mu_S(\sigma) \;=_{def}\; \{\, \langle S', \sigma' \rangle \mid [_{\epsilon \ll} S_{\ll \epsilon}] \stackrel{\sigma}{\Longrightarrow} [_{\epsilon \ll} S'_{\ll \sigma'}] \,\}$$

The *outputs* $\mu\omega$ and the *deadlock predicate* $\mu\delta$ of a pair $\langle S', \sigma' \rangle$ are defined by

1. $\mu\omega\,(\,\langle S', \sigma' \rangle\,) \;=_{def}\; \{\, x \in L_U \mid S' \stackrel{x}{\Longrightarrow} \,\}$
2. $\mu\delta\,(\,\langle S', \sigma' \rangle\,) \;=_{def}\; \sigma' = \epsilon \text{ and } \forall x \in L_U : S' \stackrel{x}{\not\Longrightarrow}$ or $\sigma' = a \cdot \sigma'' \text{ and } \forall x \in L_U \cup \{a\} : S' \stackrel{x}{\not\Longrightarrow}$

□

The reachability function serves the same purpose as $S\,\mathbf{after}\,\sigma$ (and the related expression **choice** S **after** σ) in the synchronous case. It expresses how a system proceeds after σ has been observed.

We show two properties of the reachability function. The first property, proposition 5.68, shows that $outputs_S(\sigma)$ and $\delta_S(\sigma)$ are easily obtained from $\mu_S(\sigma)$. The second one, proposition 5.69, states that the reachability function can be computed incrementally with respect to traces, i.e. $\mu_S(\sigma \cdot a)$ is easily computed from $\mu_S(\sigma)$. This means that given a set of traces $\mathcal{F} \subseteq L^*$, $outputs_S(\sigma)$ and $\delta_S(\sigma)$ can be computed systematically for all $\sigma \in \mathcal{F}$.

Proposition 5.68

1. $outputs_S(\sigma) \;=\; \bigcup \{\, \mu\omega(\,\langle S', \sigma' \rangle\,) \mid \langle S', \sigma' \rangle \in \mu_S(\sigma) \,\}$
2. $\delta_S(\sigma) \quad \text{iff} \quad \exists \langle S', \sigma' \rangle \in \mu_S(\sigma) : \mu\delta(\langle S', \sigma' \rangle)$

□

Proposition 5.69

1. $\mu_S(\epsilon) \;=\; \{\, \langle S', \epsilon \rangle \mid S \stackrel{\epsilon}{\Longrightarrow} S' \,\}$
2. $\mu_S(\sigma \cdot a) \;=\; \{\, \langle S', \sigma' \cdot a \rangle \mid \langle S', \sigma' \rangle \in \mu_S(\sigma) \,\} \cup \{\, \langle S'', \epsilon \rangle \mid \langle S', \epsilon \rangle \in \mu_S(\sigma) \text{ and } S' \stackrel{a}{\Longrightarrow} S'' \,\}$
3. $\mu_S(\sigma \cdot x) \;=\; \{\, \langle S'', \sigma'' \rangle \mid \exists \langle S', \sigma' \rangle \in \mu_S(\sigma), \exists \rho \preceq \sigma' : S' \stackrel{x \cdot \rho}{\Longrightarrow} S'' \text{ and } \sigma'' = \sigma' \backslash \rho \,\}$

□

Example 5.70
Consider S in figure 5.16. We compute $outputs_S(\sigma)$ and $\delta_S(\sigma)$ for all $\sigma \in \mathcal{F} = traces(S)$ by first computing $\mu_S(\sigma)$. The results are depicted in the tree of figure 5.20. The edges of the tree are formed by all $\sigma \in \mathcal{F} = traces(S)$; a node that is reached from the root via trace σ contains all pairs $\langle S', \sigma' \rangle \in \mu_S(\sigma)$. To compute a next node of the tree only the contents of the parent node is required. $outputs_S(\sigma)$ and $\delta_S(\sigma)$ as they follow from

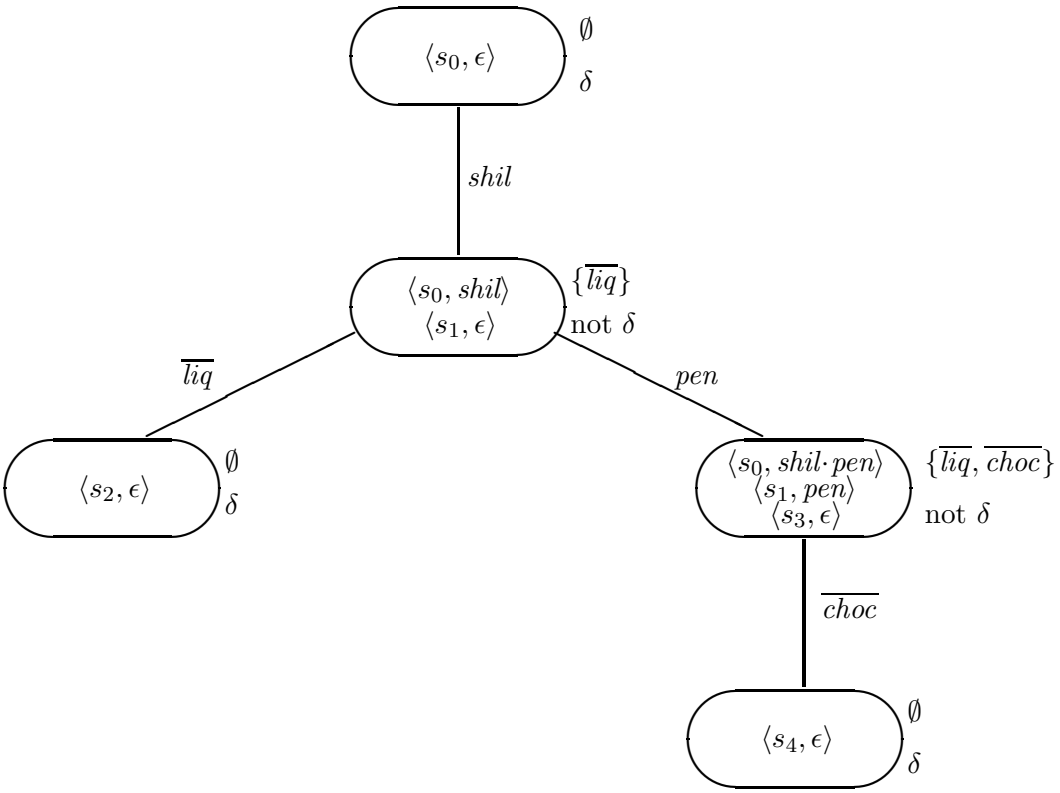


Figure 5.20: Reachability function of S (figure 5.16)

the node contents, are also depicted with each node. (cf. the failure tree of [Bri87] for the synchronous case.)

A few example computations:

$$\begin{aligned}
&\mu_S(\epsilon)\\
&\quad = \{\langle S', \epsilon\rangle \mid S \stackrel{\epsilon}{\Longrightarrow} S'\}\\
&\quad = \{\langle s_0, \epsilon\rangle\}\\
&\mu_S(shil{\cdot}\overline{liq})\\
&\quad = \{\langle S'', \sigma''\rangle \mid \exists \langle S', \sigma'\rangle \in \mu_S(shil), \exists \rho \preceq \sigma' : S' \stackrel{\overline{liq}\cdot\rho}{\Longrightarrow} S'' \text{ and } \sigma'' = \sigma' \backslash \rho\}\\
&\quad = \{\langle s_2, \epsilon\rangle\}\\
&\mu_S(shil{\cdot}pen)\\
&\quad = \{\langle S', \sigma'{\cdot}pen\rangle \mid \langle S', \sigma'\rangle \in \mu_S(shil)\}\\
&\qquad \cup \{\langle S'', \epsilon\rangle \mid \langle S', \epsilon\rangle \in \mu_S(\sigma) \text{ and } S' \stackrel{pen}{\Longrightarrow} S''\}\\
&\quad = \{\langle s_0, shil{\cdot}pen\rangle, \langle s_1, pen\rangle\} \cup \{\langle s_3, \epsilon\rangle\}\\
&outputs_S(shil{\cdot}pen)\\
&\quad = \bigcup\{\mu\omega(\langle S', \sigma'\rangle \mid \langle S', \sigma'\rangle \in \mu_S(\sigma)\}\\
&\quad = (\ out(s_0) \cup out(s_1) \cup out(s_3)\ )\lceil L_U\\
&\quad = \{\overline{liq}, \overline{choc}\}
\end{aligned}$$

□

Having the reachability function defined over labelled transition systems, the next step is to look for compositional rules to derive it from behaviour expressions in $\mathcal{BEX}$, analogous to the way it has been done for synchronous test derivation in the section 4.3. Since the reachability function contains all information needed to derive test cases, this would give a compositional way of deriving asynchronous test cases from behaviour expressions. Deriving such compositional rules is straightforward, especially when the language is restricted according to equation (4.6). In fact, the only attribute needed is B **after** σ. Elaboration of these rules is left for further work.

5.10 Concluding Remarks

This chapter presented a queue model to formalize asynchronous testing, and to relate it to the synchronous case. Many extensions of the presented theory are possible: axiomatization of queue equivalence, application to languages with asynchronous communication, other (equivalence) relations in queue contexts, other contexts, etc. We discuss a few items.

Restricted classes of specifications The presented theory of queue contexts is rather complex, especially in deriving deadlock properties of queue contexts from the original specification. Derivation of deadlocks is easier if a specification is *fully-specified* [BU91], which means that it can never refuse input actions:

$$\forall \sigma \in L^*, \forall a \in L_I :\ S \textbf{ after } \sigma \textbf{ must } L_U \cup \{a\}$$

Fully specifiedness implies that there are no blocking deadlocks (definition 5.40):

$$B\text{-}tracks(S) = \delta\text{-}block(S) = \emptyset$$

so that the complexity of finding deadlocks is reduced (proposition 5.41).

Robust systems are usually implicitly fully-specified by requiring to output an error message if a non-specified, erroneous input message occurs.

Specification languages that use asynchronous communication usually impose fully-specifiedness by some built-in mechanism. In TTCN an OTHERWISE behaviour specifies what to do if an unexpected message arrives; SDL signals that cannot be consumed by the receiving process are neglected. Note, however, that this feature of SDL complicates derivation of the tracks of a specification. For every input signal the possibility of neglecting it must be expressed explicitly in the labelled transition system by a transition that starts and ends in the same state. This implies that the set of tracks is always infinite, since every possible sequence of neglected inputs followed by an output action, constitutes a track.

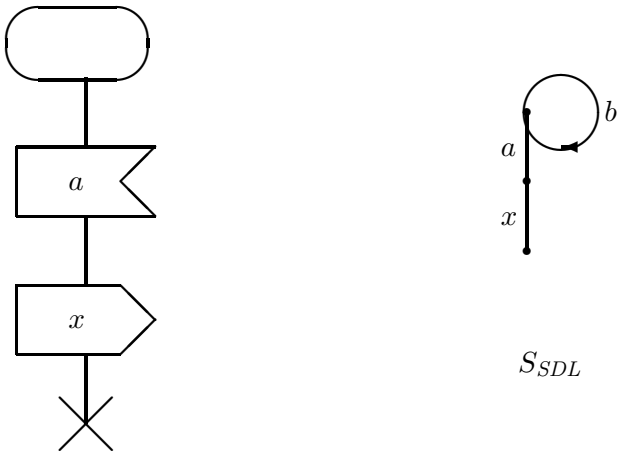


Figure 5.21: Example SDL process.

Example 5.71
The SDL specification of figure 5.21 may neglect in its initial state any input unequal to a. For $L_I = \{a, b\}$ it is represented by the transition system S_{SDL}, which has infinitely many tracks: $track(S_{SDL} = \{\epsilon,\ a{\cdot}x,\ b{\cdot}a{\cdot}x,\ b{\cdot}b{\cdot}a{\cdot}x,\ b{\cdot}b{\cdot}b{\cdot}a{\cdot}x, \ldots\}$. □

Axiomatization Given a language with labelled transition systems as underlying semantics, theorems for queue equivalence can be derived. Such theorems can improve understanding of the nature of queue equivalence, and they facilitate comparing specifications at language level. An example of such a theorem expressed in the language $\mathcal{BEX}$ is

$$x; B_1 \,\Box\, y; B_2 \quad \approx_{\mathcal{Q}} \quad \mathbf{i}; x; B_1 \,\Box\, \mathbf{i}; y; B_2$$

An interesting topic is the comparison of our work with [JJH90], where an axiomatic approach is used for embedding asynchronous communication into CSP.

Transductions Transductions are a kind of inference rules specifically suited to define the behaviour of contexts [Lar90]. The rule

$$C \xrightarrow[b]{a} C'$$

specifies that if the process in the context can do b then the whole process makes the transition a to C'. Queue contexts can be easily described using transductions. This allows to apply theories of transductions to queue contexts e.g. [Bri92], which addresses the problem of solving equations of the form $X \approx C[X]$. For queue contexts this could for instance be used to prove the intuition that queue contexts can be concatenated:

$$[_\epsilon \ll S \ll_\epsilon] \approx [_\epsilon \ll [_\epsilon \ll S \ll_\epsilon]_{\ll \epsilon}]$$

Other relations Queue contexts can be studied for other relations, e.g. bisimulation equivalence, failure trace equivalence, ready trace equivalence, etc., thus extending the diagram of proposition 5.51. In [Abr87] a hierarchy of equivalences based on synchronous testing is identified. It can be investigated how this applies to queue contexts, especially whether equivalences in the synchronous hierarchy coincide when observed in a queue context.

For test generation an important question is which implementation relations are preserved by which contexts, and whether test cases derived for one context, can be used for testing in other contexts, reducing the need to duplicate test derivation for each separate context or testing architecture.

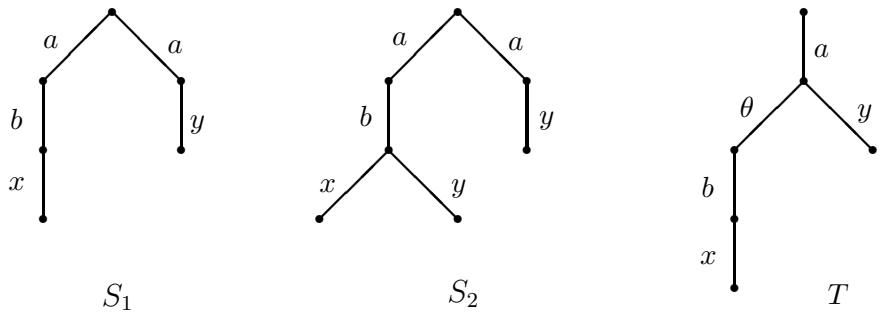


Figure 5.22: Testing with deadlock detection.

As an example consider figure 5.22: $S_1 \approx_{\mathcal{Q}} S_2$. Using the label θ as in [Lan90], with the intuitive meaning that a θ-transition can occur if and only if no other transitions

are possible, the test T can tell S_1 and S_2 apart, so they are not 'queue-failure trace' equivalent. The θ-label is able to detect deadlock, so it can be decided whether the left or right branch has been chosen. An open question is the relation between deadlock detection using θ and deadlock occurrence δ.

Potentially interesting relations can be defined by considering deadlock occurrence δ as a special 'observable' action. In many places δ already occurs like an ordinary label. By extending L_U with the special label δ, and defining traces of queue contexts over $L_I \cup L_U \cup \{\delta\}$ new (equivalence) relations based on such traces can be defined and investigated (cf. the role of deadlock δ in ACP [Bae86, BK85]).

Other contexts Many possibilities of extending the presented theory can be found in considering other contexts than the one input–one output unbounded queue. We can think of bounded queues, for example with length one (buffers), priority messages in queues, and more than one input and/or output queue. Taking more queues for one direction makes it possible that input actions sent via different queues do not arrive in the same order as they were sent, thus complicating the relation between traces of a specification and its queue context. More queues lead to an explosion of possible interleavings of orderings of send and receive of input and output actions. One might consider introducing (one of) the formalisms of *event structures*, which are better suited for coping with explosions of interleavings [Lan92].

There are infinitely many possible contexts ranging from buffers to complete services. All these contexts can be studied separately, but also within one general framework. In [BKPR91] such a framework is presented based on modelling the context (or medium) by its state space, and modelling communication by transformations performed by the communicating systems on this state space. It has to be investigated how our theory relates to that framework.

Introducing more queues in the context makes it possible to use the theory for the the problem of comparing specifications that use asynchronous communication, like Estelle and SDL, with specifications that use synchronous communication, like LOTOS. Both Estelle and SDL do allow more queues or channels to communicate with the environment. Realistic use of TTCN requires at least two pairs of input-output queues, one for the upper PCO (Point of Control and Observation) of the implementation and one for the lower PCO.

The presented theory can also be of use in deriving test cases from specifications that are based on asynchronous communication, like Estelle and SDL. For example for SDL the asynchronous communication tree (ACT) model [BH89] can be combined with the presented theory. For finite state machine specifications the generated test cases can be compared with the ones generated using finite state machine based test generation techniques like distinguishing sequences or Unique Input/Ouput sequences [ADLU88, BU91].

Test generation The test generation algorithms presented in section 5.8, using the algorithms in section 5.9, have to be extended to cope with languages like $\mathcal{BEX}$ and $\mathcal{BEX}_v$, and finally with realistic specification languages like LOTOS.

The transformation of the generated test cases to TTCN lacks a formal basis, partly because of the lack of a completely formally defined semantics for TTCN. This transformation should be elaborated and given a more formal basis, so that it can be related to the practice of conformance testing, and so that existing TTCN-based tools can be used.

Chapter 6

Test Selection

6.1 Introduction

Test generation algorithms can be used to generate a large number of test cases, given a specification in the appropriate formalism (chapters 4 and 5). All these test cases can detect errors in implementations, and errors detected with these test cases indeed indicate that an implementation does not conform to its specification. However, the number of test cases that can be generated may be very large, or even infinite. This implies that execution of all generated test cases is impossible (because the number of generated test cases is infinite), unfeasible (if the number of test cases is very large), or simply too expensive. In these cases a selection of the generated test cases has to be made. Such a reduction of the size of the (automatically) generated test suite by choosing an appropriate subset thereof is called *test selection*.

Test selection should not be done at random, but a strategy should be applied, such that the resulting reduced test suite is valuable for conformance testing, in the sense that there is a large chance of detecting non-conforming implementations. In software testing some heuristics for test selection are known, e.g. equivalence partitioning and boundary value analysis [Mye79]. In the context of protocol testing following ISO9646 (section 1.3) [ISO91a, part 2, section 10.4] gives hints for the analogous problem of selection of test purposes. (Note that the term 'test selection' in the context of ISO9646 refers to the selection process of test cases from a (standardized) abstract test suite based on values of PICS and the PIXIT (section 1.3.4).)

Test selection is an activity that in principle cannot be based solely on the formal specification of a system. In order to decide which test cases are more valuable than others, extra information, outside the realm of the specification formalism, is indispensable. Such information may include knowledge about which errors are frequently made by implementers, which kind of errors are important, e.g. in the sense of catastrophic consequences, what functionality is difficult to implement, what functionality is crucial for the well-functioning of the system, etc. Although a generally applicable method for test selection cannot be given, not even within the realm of one particular specification

formalism, still some classification and decomposition of the problem of test selection can be given.

This chapter introduces a framework for test selection, extending the ideas presented in [BAL+90]. The approach is based on assigning *values* and *costs* to test suites. Minimal requirements for assigning such costs and values are introduced. The selected test suite will be the one that combines the highest value with the lowest cost. The value assigned to a test suite is related to the error detecting capability of that test suite.

The formal framework for test selection is introduced in section 6.2, defining valuation, cost and coverage of a test suite, discussing the role of test purposes and their negations, called test failures, and introducing the weight of test purposes and test failures. A method to assign values to test suites based on the probability of rejecting non-conforming implementations is elaborated in section 6.3. In section 6.4 a simple example of a labelled transition system specification is elaborated. Finally, section 6.5 discusses a technique for test selection based on specification selection: instead of deriving a lot of test cases and selecting from them, a specification is transformed so that only a limited number of test cases is generated. This technique is elaborated for test case generation from specifications based on labelled transition systems using the implementation relation **conf**.

6.2 A Framework for Test Selection

In this section a test selection framework is presented. The presentation is independent from any specification formalism. The assumptions and notations of section 2.5 are used: specifications and implementations can be modelled by a specification formalism $\mathcal{L}_{FDT}$, conformance of implementations with respect to specifications is expressed by an implementation relation $\leq_{\mathcal{R}}$ on $\mathcal{L}_{FDT}$, tests are given in a test notation $\mathcal{L}_T$, passing a test is defined by a relation **passes** $\subseteq \mathcal{L}_{FDT} \times \mathcal{L}_T$, with the extra notation as in 2.8.

In this section we consider test selection for a given specification $S \in \mathcal{L}_{FDT}$, and we restrict test suites to a finite universe of test cases $\mathcal{K} \subseteq \mathcal{L}_T$: $\Pi \in \mathcal{P}(\mathcal{K})$. $\mathcal{K}$ is assumed to be sound with respect to S: the test cases in $\mathcal{K}$ will not reject correct implementations (definition 2.9). $\mathcal{K}$ could for instance be a complete set of test cases derived from S using a test derivation algorithm, but which is too large to be executed economically.

The test selection problem now consists of selecting a subset of $\mathcal{K}$ that is in some sense optimal. Optimality will be decomposed and formalized by introducing the *value* and the *cost* of a test suite.

6.2.1 Value Assignment

As has been argued in the introduction, test cases are selected on the basis of mostly *informal* grounds. To allow reasoning about test selection and related issues in a formal model, we must represent such knowledge separately, at a suitable level of abstraction.

We propose to do this by defining the relative weight of tests suites based on the external knowledge at the disposal of the test selector. To do this properly we must make sure that such measures for test suites are consistent with their testing capabilities. In particular, we want that test suites that reject the same set of implementations are treated as being equally interesting. The required notion of testing power is given in the following definition (cf. example 3.24).

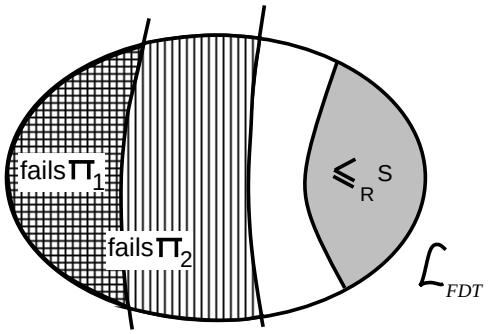


Figure 6.1: The relation $\leq_{\Pi}$: $\Pi_1 \leq_{\Pi} \Pi_2$.

Definition 6.1
For $\Pi_1, \Pi_2 \in \mathcal{P}(\mathcal{K})$ we define

$$\begin{array}{lll} \Pi_1 \leq_{\Pi} \Pi_2 & =_{def} & \{I \in \mathcal{L}_{FDT} \mid I \textbf{ fails } \Pi_1\} \subseteq \{I \in \mathcal{L}_{FDT} \mid I \textbf{ fails } \Pi_2\} \\ \Pi_1 \approx_{\Pi} \Pi_2 & =_{def} & \Pi_1 \leq_{\Pi} \Pi_2 \text{ and } \Pi_2 \leq_{\Pi} \Pi_1 \end{array}$$

When $\Pi_1 \leq_{\Pi} (\approx_{\Pi}) \Pi_2$ holds, we say that Π_1 has less (equal) testing power than (as) Π_2 (in $\mathcal{L}_{FDT}$).

□

The testing power of test suites is compared by comparing their sets of detected, erroneous implementations $\{I \in \mathcal{L}_{FDT} \mid I \textbf{ fails } \Pi\}$: the larger this set the more powerful a test suite is (figure 6.1).

Having introduced a way for comparing the testing power of test suites, we can define their *value* by way of *valuations*, i.e. mappings from test suites to weights, that respect the preorder $\leq_{\Pi}$.

Definition 6.2
A *valuation* of $\mathcal{P}(\mathcal{K})$ is a function $v : \mathcal{P}(\mathcal{K}) \to \mathbf{R}_{\geq 0}$ such that for all $\Pi_1, \Pi_2 \in \mathcal{P}(\mathcal{K})$

$$\Pi_1 \leq_\Pi \Pi_2 \quad \text{implies} \quad v(\Pi_1) \leq v(\Pi_2) \tag{6.1}$$

□

It follows that for all valuations v and test suites $\Pi_1, \Pi_2 \in \mathcal{P}(\mathcal{K})$ if $\Pi_1 \approx_\Pi \Pi_2$ then $v(\Pi_1) = v(\Pi_2)$.

The choice of the set of weights in definition 6.2, the set of non-negative reals, is merely a convenient one, and we could have chosen another, e.g. the set of natural numbers.

The first question to be asked now, of course, is how to define suitable valuations. This question has two components. First, how does one establish the proper value $v(\Pi)$ given a $\Pi \in \mathcal{P}(\mathcal{K})$, and second, how to make sure that property (6.1) is fulfilled. We concentrate on the latter aspect first. The first aspect is discussed in the next two sections.

Having related the testing power of a test suite to its set of detected, erroneous implementations, it is natural to relate also the value of a test suite to such sets.

Definition 6.3
A valuation of $\mathcal{P}(\mathcal{L}_{FDT})$ is a function $v : \mathcal{P}(\mathcal{L}_{FDT}) \to \mathbf{R}_{\geq 0}$ such that for all $c_1, c_2 \in \mathcal{P}(\mathcal{L}_{FDT})$

$$c_1 \subseteq c_2 \quad \text{implies} \quad v(c_1) \leq v(c_2) \tag{6.2}$$

□

Proposition 6.4
If $v : \mathcal{P}(\mathcal{L}_{FDT}) \to \mathbf{R}_{\geq 0}$ is a valuation of $\mathcal{P}(\mathcal{L}_{FDT})$, then $\overline{v} : \mathcal{P}(\mathcal{K}) \to \mathbf{R}_{\geq 0}$, defined by

$$\overline{v}(\Pi) \quad =_{def} \quad v\,(\,\{\, I \in \mathcal{L}_{FDT} \mid I \textbf{ fails } \Pi \,\}\,)$$

is a valuation of $\mathcal{P}(\mathcal{K})$.

□

Now that we have reallocated the problem of finding valuations, we must study ways to find and define such mappings for sets of (erroneous) implementations. We study three methods: representative error cases, error equivalence, and test purposes.

Representative Error Cases

One way to ensure the validity of (6.2), is to define v in the following manner. Let $\{e_1, \ldots, e_n\} \subseteq \mathcal{L}_{FDT}$ be a set containing specifications of typical *error cases* (figure 6.2), and assign to each a weight $w(e_i)$ $(1 \leq i \leq n)$ according to its gravity. The value of a subset of $\mathcal{L}_{FDT}$ can then be defined as the sum of the weights of all error cases that are contained in it, or, equivalently, the value of a test suite Π is the sum of the weights of all error cases that fail it.

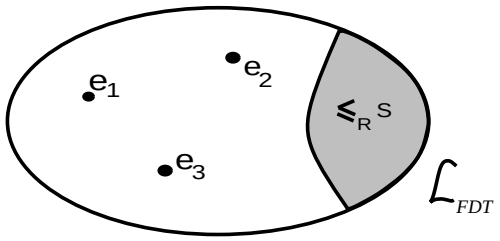


Figure 6.2: Representative error cases.

Definition 6.5
Let $E \subseteq \mathcal{L}_{FDT}$ be finite, and $w : E \to \mathbf{R}_{\geq 0}$ then $w_r : \mathcal{P}(\mathcal{L}_{FDT}) \to \mathbf{R}_{\geq 0}$ is defined by

$$w_r(c) \quad =_{def} \quad \Sigma\{w(e) \mid e \in E \text{ and } e \in c\}$$

□

Proposition 6.6
For all finite $E \subseteq \mathcal{L}_{FDT}$ and $w : E \to \mathbf{R}_{\geq 0}$, w_r is a valuation of $\mathcal{P}(\mathcal{L}_{FDT})$, and $\overline{w_r}$ is a valuation of $\mathcal{P}(\mathcal{K})$.

□

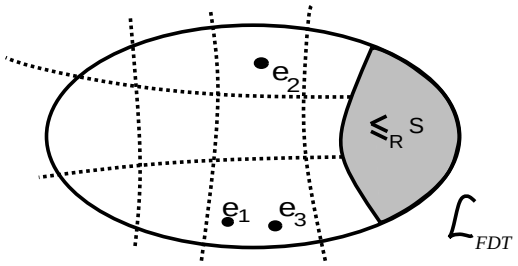


Figure 6.3: Error cases that are not representative.

The above method provides a rather practical means for establishing valuations, but merits some further investigation as it may produce unintended results, when not used carefully. First of all, we must establish that the error cases are in some sense representative of all the classes of errors that can occur in an implementation. Secondly, we must make sure that some classes of errors are not unintentionally overrepresented, i.e. different error cases should represent different error classes. As the total effective weight assigned to an error class is the sum of the weights of the represented error cases in that class, having more than one error case out of an error class could produce misleading results (figure 6.3).

Error Equivalence

To obtain a more homogeneous valuation of $\mathcal{P}(\mathcal{L}_{FDT})$ we can define a partition over $\mathcal{P}(\mathcal{L}_{FDT})$, so that from each partition class containing erroneous implementations exactly one error case is selected. By choosing a suitable partitioning of $\mathcal{P}(\mathcal{L}_{FDT})$ a better spread of error cases is obtained.

But as the error cases were only introduced as a way to define valuations, we can skip them and use the error classes themselves, i.e. assign weights to partition classes. To do so we must be sure that the subsets of $\mathcal{L}_{FDT}$ for which we wish to define a valuation, are *expressible* in the partition, i.e. these subsets must be expressible as unions of partition classes. The value of $p \in \mathcal{P}(\mathcal{L}_{FDT})$ can then be defined as the sum of the partition classes that form p.

Definition 6.7
Let $\Xi = \{\xi_1, \ldots, \xi_n\}$ be a finite partition of $\mathcal{L}_{FDT}$, then

1. $c \subseteq \mathcal{L}_{FDT}$ is *expressible* in Ξ, if $c = \bigcup\{\xi_i \in \Xi \mid \xi_i \subseteq c\}$.
2. The set of subsets of $\mathcal{L}_{FDT}$ that are expressible in Ξ is denoted by $\mathcal{E}(\Xi)$:

$$\mathcal{E}(\Xi) \quad =_{def} \quad \{\ c \subseteq \mathcal{L}_{FDT} \mid c = \bigcup\{\xi_i \in \Xi \mid \xi_i \subseteq c\}\ \}$$

3. Let $w : \Xi \to \mathbf{R}_{\geq 0}$, then $w_p : \mathcal{E}(\Xi) \to \mathbf{R}_{\geq 0}$ is defined by

$$w_p(c) \quad =_{def} \quad \Sigma\{w(\xi_i) \mid \xi_i \subseteq c\}$$

□

Proposition 6.8
For all $w : \Xi \to \mathbf{R}_{\geq 0}$, w_p is a valuation of $\mathcal{E}(\Xi)$.

□

The first natural partitioning of $\mathcal{L}_{FDT}$ is induced by an equivalence relation based on $\mathcal{K}$. The idea is that partition classes are formed by error cases that cannot be distinguished by any test suite in $\mathcal{P}(\mathcal{K})$.

Definition 6.9
For $I_1, I_2 \in \mathcal{L}_{FDT}$ we define

$$I_1 \approx_{\mathcal{K}} I_2 \quad =_{def} \quad \forall \Pi \in \mathcal{P}(\mathcal{K}) : I_1 \textbf{ fails } \Pi \quad \text{iff} \quad I_2 \textbf{ fails } \Pi$$

□

It is clear that $\approx_{\mathcal{K}}$ is an equivalence relation over $\mathcal{L}_{FDT}$, and that the partition $\mathcal{L}_{FDT}/\approx_{\mathcal{K}}$ is finite if $\mathcal{K}$ is finite. The relation $\approx_{\mathcal{K}}$ is related to the notion of testing equivalence ((3.1) in section 3.2): two processes are equivalent if they cannot be distinguished by any test. The difference between $\approx_{\mathcal{K}}$ and $\approx_T$ is the class of test cases: $\approx_{\mathcal{K}}$ depends on a specification S, hence for each S another equivalence is defined. For $\approx_{\mathcal{K}}$ the equivalence class consisting of processes that pass all tests, contains all correct implementations of

S, due to our assumption that $\mathcal{K}$ is sound. This class will normally be assigned value 0: rejection of implementations from this class is of no interest at all, on the contrary.

For any Π the set $\{I \in \mathcal{L}_{FDT} \mid I \textbf{ fails } \Pi\}$ is expressible in the partition $\mathcal{L}_{FDT}/\approx_{\mathcal{K}}$, as follows from the fact that $\mathcal{L}_{FDT}/\approx_{\mathcal{K}}$ gives the maximal resolution power that can be realized with test suites in $\mathcal{P}(\mathcal{K})$. Hence the following proposition.

Proposition 6.10
For all $u : \mathcal{L}_{FDT}/\approx_{\mathcal{K}} \rightarrow \mathbf{R}_{\geq 0}$, $\overline{u_p} : \mathcal{P}(\mathcal{K}) \rightarrow \mathbf{R}_{\geq 0}$ is a valuation of $\mathcal{P}(\mathcal{K})$.

□

Any valuation that is based on assigning weights to error cases can be defined as a valuation based on assigning weights to the equivalence classes of $\approx_{\mathcal{K}}$.

Proposition 6.11
For all finite $E \subseteq \mathcal{L}_{FDT}$ and $w : E \rightarrow \mathbf{R}_{\geq 0}$ there exists a $u : \mathcal{L}_{FDT}/\approx_{\mathcal{K}} \rightarrow \mathbf{R}_{\geq 0}$ such that $\overline{u_p} = \overline{w_r}$.

□

Test Purposes

By considering error classes in $\mathcal{L}_{FDT}/\approx_{\mathcal{K}}$ one obviously considers the maximal power of resolution that can be realized with test suites in $\mathcal{P}(\mathcal{K})$. In a given case, however, one will only require resolution in specific areas of $\mathcal{L}_{FDT}$, whereas one may be quite willing to forget about possible distinctions that could be made in other places. If we somehow want to take into account only those distinctions that are felt to be relevant in a given situation we enter the domain of the *test purposes*. Informally, a test purpose is a statement explaining the intention of the test case or test suite at hand (section 1.3.3). We formalized it in section 2.2.3 as a requirement or predicate, for which a valid test case or test suite is to be developed. Such a test purpose p can be identified with the set of all implementations satisfying it:

$$p \;=\; \{\; I \in \mathcal{L}_{FDT} \mid I \textbf{ sat } p \;\} \;\subseteq\; \mathcal{L}_{FDT} \tag{6.3}$$

If p is a test purpose, i.e. a requirement that a correct implementation should satisfy, then its negation expresses a failure. We call it a *test failure*, and denote it by f_p:

$$f_p \;=\; \{\; I \in \mathcal{L}_{FDT} \mid I \textbf{ sat} \not{} p \;\} \;\subseteq\; \mathcal{L}_{FDT} \tag{6.4}$$

Let a set of test purposes $P = \{p_1, \ldots, p_n\}$ be given, then P defines an equivalence on $\mathcal{L}_{FDT}$:

Definition 6.12
For $I_1, I_2 \in \mathcal{L}_{FDT}$ we define

$$I_1 \approx_P I_2 \quad =_{def} \quad \forall p \in P : I_1 \textbf{ sat } p \;\; \text{iff} \;\; I_2 \textbf{ sat } p$$

□

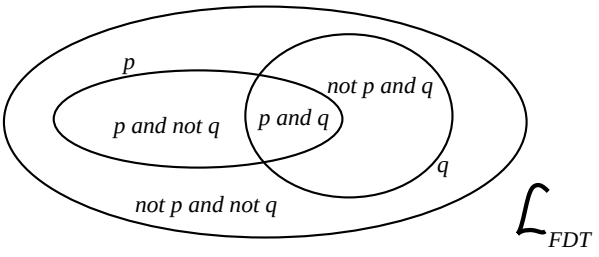


Figure 6.4: Test purposes p and q.

It is clear that also $\approx_P$ is an equivalence relation over $\mathcal{L}_{FDT}$, and that the partition $\mathcal{L}_{FDT}/\approx_P$ is finite if P is finite.

The partition $\mathcal{L}_{FDT}/\approx_P$ consists of at most 2^n classes, one of which is the class of implementations that satisfy all test purposes. All other classes contain erroneous implementations: they have one or more of the failures in $F = \{f_1, \ldots, f_n\} = \{\text{not}p_1, \ldots, \text{not}p_n\}$. Figure 6.4 gives an example for $P = \{p, q\}$: $\mathcal{L}_{FDT}$ is partitioned into 4 classes; the class (p and q) contains the correct implementations, implementations in (not p and q) have failure f_p while they satisfy q, implementations in (p and not q) have one failure f_q, and the class (not p and not q) contains implementations with two errors.

If we assign weights to each possible combination of failures, i.e. to each equivalence class of $\mathcal{L}_{FDT}/\approx_P$, a value assignment for expressible subsets of $\mathcal{L}_{FDT}$ follows immediately from propositions 6.8 and 6.4.

Proposition 6.13
Let $u : \mathcal{L}_{FDT}/\approx_P \rightarrow \mathbf{R}_{\geq 0}$, then

1. $u_p : \mathcal{E}(\mathcal{L}_{FDT}/\approx_P) \rightarrow \mathbf{R}_{\geq 0}$ is a valuation of $\mathcal{E}(\mathcal{L}_{FDT}/\approx_P)$;
2. Let E be the set of test suites for which the set of detected implementations is expressible in $\mathcal{L}_{FDT}/\approx_P$: $E = \{\Pi \subseteq \mathcal{K} \mid \{I \mid I \textbf{ fails } \Pi\} \in \mathcal{E}(\mathcal{L}_{FDT}/\approx_P)\}$, then $\overline{u_p} : E \rightarrow \mathbf{R}_{\geq 0}$ is a valuation for E.

□

Combining the definitions and propositions we come to a method for assigning values, based on test purposes, to test suites Π of which the set of detected erroneous implementations is expressible in the partition induced by a given set of relevant test purposes $\{p_1, \ldots, p_n\}$:

1. assign a weight expressing the value of the rejection of each exhaustive combination of failures, i.e. define a $u : \mathcal{L}_{FDT}/\approx_P \rightarrow \mathbf{R}_{\geq 0}$;
2. determine u_p following definition 6.7;
3. according to proposition 6.13 we can take $\overline{u_p}$ as a valuation for each Π with $\{I \mid I \textbf{ fails } \Pi\}$ expressible in $\mathcal{L}_{FDT}/\approx_P$.

Coverage

The coverage of a test suite can be seen as the extent to which the purpose of testing is approximated. Having related the value of a test suite Π to the subset of detected, erroneous implementations, it is the extent with which this set 'covers' the set of all erroneous implementations (figure 6.5). It is, in other words, the normalization of valuations: it is the ratio between the detected failures (by Π) and the possible failures (according to test purposes P)

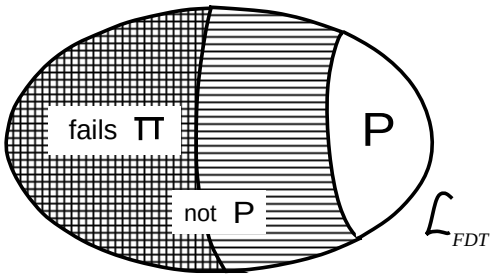


Figure 6.5: The coverage of a test suite Π.

Definition 6.14
Let P be a finite set of test purposes, and $v : \mathcal{P}(\mathcal{L}_{FDT}) \to \mathbf{R}_{\geq 0}$ a valuation. For $\Pi \in \mathcal{P}(\mathcal{K})$ we define the *coverage* of P by Π under v (or $\overline{v}$, see proposition 6.4) as

$$cov_P^v(\Pi) \quad =_{def} \quad \frac{v(\{I \in \mathcal{L}_{FDT} \mid I \textbf{ fails } \Pi\})}{v(\{I \in \mathcal{L}_{FDT} \mid I \textbf{ sa}\not{\textbf{t}} P\})} \quad = \quad \frac{\overline{v}(\Pi)}{v(\{I \in \mathcal{L}_{FDT} \mid I \textbf{ sa}\not{\textbf{t}} P\})}$$

If the denominator equals 0, i.e. if rejection of an implementation not fulfilling the test purpose P is of no value, then we define $cov_P^v(\Pi) = 1$ for every Π. If the valuation v is known from the context, or is irrelevant we write simply $cov_P(\Pi)$.

□

6.2.2 Cost Assignment

The final ingredient that we introduce is a notion of *cost* for test suites. This is quite straightforward: our only requirement is that cost increases with size.

Definition 6.15
A *cost assignment* of $\mathcal{P}(\mathcal{K})$ is a function $c : \mathcal{P}(\mathcal{K}) \to \mathbf{R}_{\geq 0}$ such that for all $\Pi_1, \Pi_2 \in \mathcal{P}(\mathcal{K})$

$$\Pi_1 \subseteq \Pi_2 \quad \text{implies} \quad c(\Pi_1) \leq c(\Pi_2)$$

□

The requirement in definition 6.15 is easily satisfied. It suffices, for example, to define c on the basis of a cost function $k : \mathcal{K} \to \mathbf{R}_{\geq 0}$ by putting

$$c(\Pi) = \Sigma\{k(t) \mid t \in \Pi\} \tag{6.5}$$

The reason that we do not take (6.5) as the basis of the definition is that the cost of executing a test case in a test suite Π may in general depend on the other test cases with which it is combined.

Having come at the end of this section, we may now formulate the formal problem of *test selection* for a given set of test purposes P as selecting $\Pi \in \mathcal{P}(\mathcal{K})$ maximizing $cov_P(\Pi)$ while minimizing $c(\Pi)$. Such optimization problems can, of course, be formulated under further constraints, such as the requirement that $cov_P(\Pi)$ must attain at least some minimal value and/or $c(\Pi)$ must not exceed some maximal cost.

6.3 Probabilities as Valuations

In the previous section the basic requirements for valuations have been identified, and it was indicated how to relate valuations to failures that a test suite detects. In this section a method to define a valuation is given that fulfils these requirements, as an application of the formal framework in the previous section. The method works from the intuition that identifies the value of a test suite with the probability that a non-conforming implementation is detected. This probability is proportional to the probability of occurrence of the failures that are detected by that test suite. This leads to defining the value of a test suite as the probability that the test failures that it detects, occur.

Consider the occurrence of an implementation I as a stochastic experiment with possible outcome in $\mathcal{L}_{FDT}$. Let $c \subseteq \mathcal{L}_{FDT}$, then by

$$Pr[I \in c]$$

we denote the probability that an arbitrary implementation $I \in c$. In the same way

$$Pr[I \in \{I \mid I \textbf{ fails } \Pi\}] = Pr[I \textbf{ fails } \Pi]$$

is the probability that test suite Π detects an erroneous implementation. This probability can be used in a straightforward way to define the value of a test suite:

$$v(\Pi) = Pr[I \textbf{ fails } \Pi] \tag{6.6}$$

The function v thus defined is a valuation:

Suppose	$\Pi_1 \leq_{\Pi} \Pi_2$
then	$\{I \in \mathcal{L}_{FDT} \mid I \textbf{ fails } \Pi_1\} \subseteq \{I \in \mathcal{L}_{FDT} \mid I \textbf{ fails } \Pi_2\}$
implies	$Pr[I \in \{I \in \mathcal{L}_{FDT} \mid I \textbf{ fails } \Pi_1\}] \leq Pr[I \in \{I \in \mathcal{L}_{FDT} \mid I \textbf{ fails } \Pi_2\}]$
implies	$v(\Pi_1) \leq v(\Pi_2)$

The coverage of a test suite Π with respect to a test purpose P in a probabilistic setting can be given intuitively as: the probability that a failing implementation will be detected by Π. This can be formalized by the conditional probability

$$\begin{array}{rl} & Pr[I \textbf{ fails } \Pi \mid I \textbf{ sa}\not{\textbf{t}} \ P] \\ = & \dfrac{Pr[I \textbf{ fails } \Pi \text{ and } I \textbf{ sa}\not{\textbf{t}} \ P]}{Pr[I \textbf{ sa}\not{\textbf{t}} \ P]} \\ = & (* \text{ using soundness of } \Pi \subseteq \mathcal{K} \ *) \\ & \dfrac{Pr[I \textbf{ fails } \Pi]}{Pr[I \textbf{ sa}\not{\textbf{t}} \ P]} \end{array}$$

This is equal to the coverage according to the previous section (definition 6.14):

$$cov_P(\Pi) \ = \ \frac{v(\{I \in \mathcal{L}_{FDT} \mid I \textbf{ fails } \Pi\})}{v(\{I \in \mathcal{L}_{FDT} \mid I \textbf{ sa}\not{\textbf{t}} \ P\})} \ = \ \frac{Pr[I \textbf{ fails } \Pi]}{Pr[I \textbf{ sa}\not{\textbf{t}} \ P]}$$

6.4 Elaborated Example

In this section the framework of section 6.2 and the valuation function of section 6.3 are applied to test suites generated from a simple labelled transition system specification.

For $\mathcal{L}_{FDT}$ we have the class of labelled transition systems $\mathcal{LTS}$ (section 1.4); for the implementation relation **conf** is chosen (definition 3.12); **passes** is defined by definition 3.25. The predicates **after** σ **must** A and **after** σ **refuses** A over $\mathcal{LTS}$ are used to express test purposes and test failures (definition 3.4).

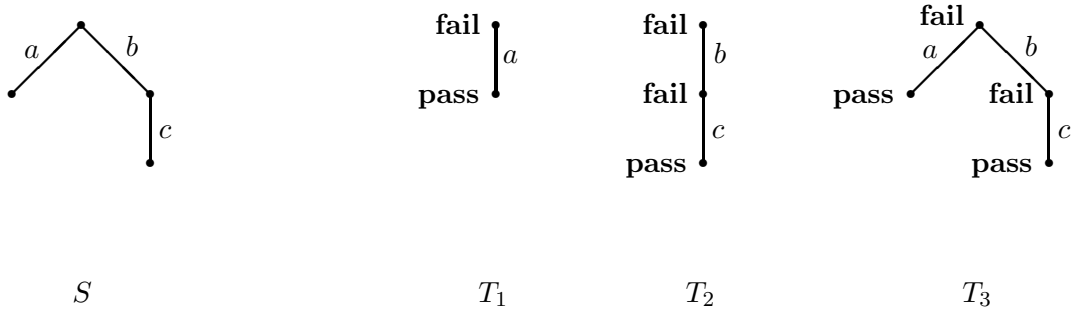


Figure 6.6: Specification S and test cases in $\mathcal{K}$.

We consider the specification S in figure 6.6 with the test cases $\mathcal{K} = \{T_1, T_2, T_3\}$. These test cases can for instance be obtained using algorithm 4.11. The test suite $\mathcal{K}$ is complete for S with respect to **conf**, and $\mathcal{K}$ contains only three elements, so we could execute them all. However, suppose we are allowed, e.g. due to cost constraints, to execute only one test case; which one do we choose?

In order to compare the testing power of these test cases we will discuss three possible valuation functions: a valuation based on representative error cases, a valuation based on the equivalence $\approx_{\mathcal{K}}$, and a valuation based on test purposes and the probability that a non-conforming implementation is detected.

Representative Error Cases

The first possibility for a valuation mentioned in the previous section is to choose representative error cases $e_1, \ldots, e_n$, and to assign weights to these error cases. A valuation can be defined as in proposition 6.4 and definition 6.5. We choose the error cases as in figure 6.7.

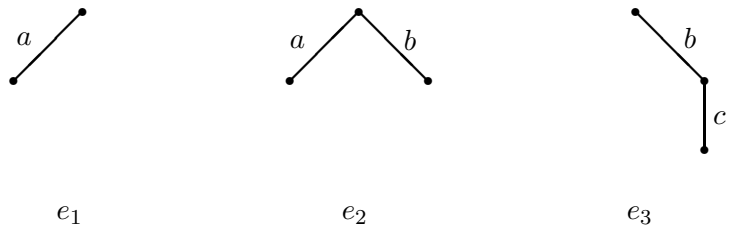


Figure 6.7: Representative error cases.

If we assign weights to e_1, e_2, e_3 according to the number of 'forgotten' transitions, $w(e_1) = 2$, $w(e_2) = 1$, $w(e_3) = 1$, we obviously get:

$$\begin{aligned} v(\{T_1\}) &= w(e_3) = 1 \\ v(\{T_2\}) &= w(e_1) + w(e_2) = 2 + 1 = 3 \\ v(\{T_3\}) &= w(e_2) = 1 \\ v(\{T_1, T_2\}) &= w(e_1) + w(e_2) + w(e_3) = 2 + 1 + 1 = 4 \\ v(\{T_1, T_2, T_3\}) &= w(e_1) + w(e_2) + w(e_3) = 2 + 1 + 1 = 4 \end{aligned}$$

So, according to this valuation, T_3 is superfluous: $\{T_1, T_2\}$ has the same value as the maximal possible value $v(\{T_1, T_2, T_3\})$. If the costs of executing a test case would allow us to execute only one test case, T_2 is the preferred one.

Error Equivalence

The finest possible distinctions between different non-conforming implementations are given by the equivalence $\approx_{\mathcal{K}}$. After some calculation 6 equivalence classes are found, one of which is the class of correct implementations (c_0). Representatives of each class are given in figure 6.8. The sets given in the nodes in these trees represent failures: these sets of actions are refused by the implementation (I **after** σ **refuses** A) while they may not be refused according to the specification (S **after** σ **must** A). According to

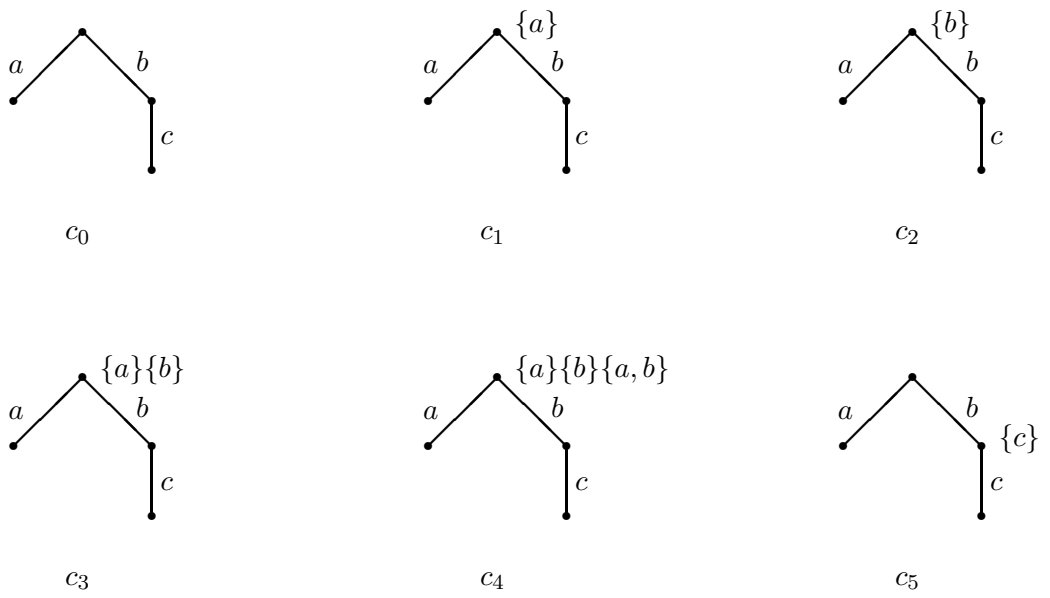


Figure 6.8: $\approx_{\mathcal{K}}$ equivalence classes.

the definition of **conf** (definition 3.12) such a combination of σ and A is a failure in the implementation.

The equivalence classes rejected by the test cases T_1, T_2, T_3 are:

T_1	detects :	c_1		c_3	c_4	
T_2	detects :		c_2	c_3	c_4	c_5
T_3	detects :				c_4	c_5

If we assign equal weight to all detected classes c_i, $w(c_i) = 1$ for $1 \leq i \leq 5$, we get:

$$\begin{array}{lcl} v(\{T_1\}) & = & 3 \\ v(\{T_2\}) & = & 4 \\ v(\{T_3\}) & = & 2 \\ v(\{T_1, T_2\}) & = & 5 \\ v(\{T_1, T_3\}) & = & 4 \\ v(\{T_1, T_2, T_3\}) & = & 5 \end{array}$$

Also with this valuation $v(\{T_1, T_2\})$ is equal to $v(\{T_1, T_2, T_3\})$; test case T_3 does not contribute to the value of this test suite. Note that T_3 does have influence on $\approx_{\mathcal{K}}$: it discriminates between c_3 and c_4.

More sophisticated assignments of weights to equivalence classes are possible, e.g. counting the number of erroneous refusals in the different error classes:
$w(c_1) = 1$ ($\{a\}$ is refused), ..., $w(c_4) = 3$ ($\{a\}, \{b\}, \{a, b\}$ are refused), etc.

Probabilistic Valuation

A more sophisticated way to assign values to test suites is by taking the probability that an implementation is rejected, as explained in section 6.3.

In order to assign probabilities to the occurrence of implementations we determine the probability for occurrence of particular failures. The probability for occurrence of a particular implementation is equal to the probability that a particular combination of failures occurs.

If we have an implementation in which n failures, $f_1, \ldots, f_n$, can occur, with respective probabilities $\alpha_1, \ldots, \alpha_n$, then the probability that an implementation has the failures f_1 and f_2, and no other failures is:

$$\alpha_1 \cdot \alpha_2 \cdot (1 - \alpha_3) \cdot \ldots \cdot (1 - \alpha_n) \tag{6.7}$$

From probability theory (e.g. [Tri82]) it is known that this is only valid if the failures are independent. Two failures are independent if they can occur independently in implementations, i.e. if one of them does not imply the other. This can be related to logical independence of a set of failures, defined in (2.28) in section 2.4.4: a set of failures can be considered independent if the corresponding set of test purposes is independent.

Remark 6.16
In fact, logical independence is a minimal requirement for independence of probabilities. Using logical independence we only require that for each possible failure an implementation can be made that has exactly that failure and no other failures. We neglect that in the implementation process logically independent failures are usually related, e.g. it is more likely that the addition $34 + 28$ is incorrectly implemented, if $34 + 27$ is also incorrectly implemented. However, we consider these possible failures independent, since it is possible that $34 + 27$ is implemented incorrectly, while $34 + 28$ is correctly implemented.

□

For labelled transition systems with **conf**, requirements, and thus also test purposes, are expressed in $\mathcal{L}_{must}$ (section 3.4.1). This implies that test failures, negations of test purposes, are expressed in $\mathcal{L}_{ref}$ (proposition 3.5.1). Logical independence for sets of requirements in $\mathcal{L}_{must}$ was investigated in proposition 3.32.

Now considering our example specification S of figure 6.6 we can derive the following set of test purposes P_S. Note that in this case the set of test purposes is complete, i.e. I **conf** S if and only if I **sat** p_i for $i = 1, 2, 3$.

$$\begin{array}{rcll} P_S & = & \{ & \textbf{after } \epsilon \textbf{ must } \{a\} & (p_1) \\ & & & \textbf{after } \epsilon \textbf{ must } \{b\} & (p_2) \\ & & & \textbf{after } b \textbf{ must } \{c\} \ \} & (p_3) \end{array}$$

The corresponding failures are the negations of the test purposes.

$$\begin{array}{rcll} F_S & = & \{ & \textbf{after } \epsilon \textbf{ refuses } \{a\} \quad (f_1) \\ & & & \textbf{after } \epsilon \textbf{ refuses } \{b\} \quad (f_2) \\ & & & \textbf{after } b \textbf{ refuses } \{c\} \; \} \quad (f_3) \end{array}$$

It follows easily from proposition 3.32 that these failures are independent. So, analogous to (6.7), the probability that an implementation has failures f_1 and f_3 and satisfies test purpose P_2 is

$$Pr[I \textbf{ sat } \{f_1, p_2, f_3\}] = \alpha_1 \cdot (1 - \alpha_2) \cdot \alpha_3$$

Considering test purposes and test failures as sets ((6.3) and (6.4)), $f_1 \cap p_2 \cap f_3$ is an element of $\mathcal{LTS}/\approx_{P_S}$. Analogously we can calculate weights for the other elements of $\mathcal{LTS}/\approx_{P_S}$. It follows that we have a weight assignment for $\mathcal{LTS}/\approx_{P_S}$, which, according to proposition 6.13.1, can be extended to a valuation for subsets of $\mathcal{LTS}$ that are expressible in $\mathcal{LTS}/\approx_{P_S}$. And following proposition 6.13.2 we have a valuation for test suites. In figure 6.9 weights w for the elements of $\mathcal{LTS}/\approx_{P_S}$ are given, where $\alpha_1 = \alpha_2 = \alpha_3 = 0.1$.

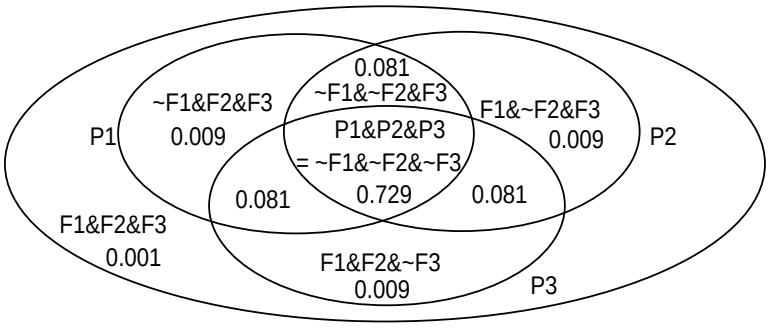


Figure 6.9: Weight assignment w for $\mathcal{LTS}/\approx_{P_S}$

To calculate the value for the test suite $\{T_1\}$ we have

$$\begin{array}{rl} & v\,(\,\{T_1\}\,) \\ = & (*\ \text{proposition 6.13.2}\ *) \\ & \overline{w_p}\,(\,\{T_1\}\,) \\ = & (*\ \text{proposition 6.4}\ *) \\ & w_p\,(\,\{I \mid I \textbf{ fails } \{T_1\}\}\,) \\ = & (*\ \text{definition 6.7.3}\ *) \\ & \Sigma\{\, w(\xi_i) \mid \xi_i \subseteq \{I \mid I \textbf{ fails } \{T_1\}\}\,\} \\ = & w(f_1 \cup f_2 \cup f_3) + w(f_1 \cup f_2 \cup p_3) + w(f_1 \cup p_2 \cup f_3) + w(f_1 \cup p_2 \cup p_3) \\ = & \quad Pr[I \textbf{ sat } \{f_1, f_2, f_3\}] + Pr[I \textbf{ sat } \{f_1, f_2, p_3\}] \\ & + Pr[I \textbf{ sat } \{f_1, p_2, f_3\}] + Pr[I \textbf{ sat } \{f_1, p_2, p_3\}] \\ = & (*\ \text{figure 6.9}\ *) \\ & 0.001 + 0.009 + 0.009 + 0.081 \\ = & 0.100 \end{array}$$

Π	$\{I \mid I$ **fails** $\Pi\}$	$v(\Pi)$	$cov_{P_S}(\Pi) = \frac{v(\Pi)}{1-Pr[I \textbf{ sat } P_S]}$
$\{T_1\}$	f_1	0.100	0.369
$\{T_2\}$	$f_2 \cup f_3$	0.190	0.701
$\{T_3\}$	f_3	0.100	0.369
$\{T_1, T_2\}$	$f_1 \cup f_2 \cup f_3$	0.271	1.000
$\{T_1, T_3\}$	$f_1 \cup f_3$	0.190	0.701
$\{T_1, T_2, T_3\}$	$f_1 \cup f_2 \cup f_3$	0.271	1.000

Table 6.1: Value and coverage of example test suites.

The values and coverages of some test suites are given in table 6.1. Also for this valuation T_2 is the most powerful test case, and the suite $\{T_1, T_2\}$ has the same testing power as $\{T_1, T_2, T_3\}$. An interesting exercise would be to vary the values of α_i with $\Delta\alpha$, in order to see how these qualitative results depend on the values chosen for α_i. This analysis is left to the reader.

6.5 Test Selection by Specification Selection

Test selection was introduced in section 6.1 as finding a subset of an (automatically) generated set of test cases that is in some sense optimal. The previous sections mainly dealt with formalizing optimality. This section discusses techniques for finding subsets of test suites: given S, $\leq_{\mathcal{R}}$, and $\Pi_{\mathcal{R}}$, determine a $\Pi' \subseteq \Pi_{\mathcal{R}}(S)$ (figure 6.10). The obvious two step technique of first generating all possible test cases and then making a selection from this set, may be impossible (if the generated test suite is infinite), and is at least undesirable. Instead of generating too many test cases, and selecting from them, one could try to avoid this overproduction, i.e. deriving Π' directly. This can be done by taking another test derivation algorithm $\Pi'_{\mathcal{R}}$, such that $\Pi'_{\mathcal{R}}(S) \subseteq \Pi_{\mathcal{R}}(S)$, or by transforming S into S', such that $\Pi_{\mathcal{R}}(S') \subseteq \Pi_{\mathcal{R}}(S)$ (figure 6.10). If $\Pi_{\mathcal{R}}$ is sound, a necessary requirement for correctness of $\Pi'_{\mathcal{R}}$ and S' is soundness of $\Pi'_{\mathcal{R}}(S)$ and $\Pi_{\mathcal{R}}(S')$ respectively.

An easy way to define a test derivation $\Pi'_{\mathcal{R}}$ is by taking another implementation relation $\leq_{\mathcal{R}'}$ with corresponding sound test derivation $\Pi_{\mathcal{R}'}$. It is easily checked that if the relation $\leq_{\mathcal{R}'}$ is weaker than $\leq_{\mathcal{R}}$ the soundness criterion is fulfilled.

Proposition 6.17
Let $\Pi_{\mathcal{R}}$ and $\Pi_{\mathcal{R}'}$ be sound test derivations for the implementation relations $\leq_{\mathcal{R}}$ and $\leq_{\mathcal{R}'}$ respectively. If $\leq_{\mathcal{R}} \subseteq \leq'_{\mathcal{R}}$ then $\Pi_{\mathcal{R}'}$ is sound for $\leq_{\mathcal{R}}$.

□

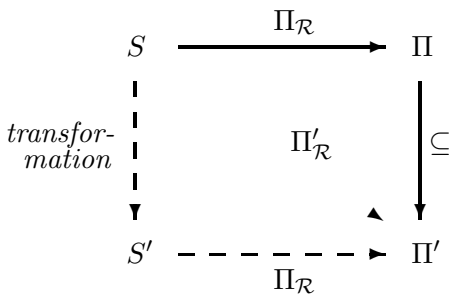


Figure 6.10: Test selection.

In this way a **conf** test suite can be seen as a selection of a $\leq_{te}$ test suite (chapters 3 and 4), and an $\mathbf{asco}, \mathbf{aconf}, \mathbf{aconf}^n, \leq_{tr(S)}$, or $\leq_{tr(Q_S)}$ test suite as a selection of a $\leq_{\mathcal{O}}$ test suite (chapter 5, proposition 5.55.1).

To describe the transformation of a specification S into S' with the purpose of doing test selection we introduce a *selection transformation* $\Theta_{\mathcal{R}}$.

Definition 6.18
Let $\leq_{\mathcal{R}}$ be an implementation relation on $\mathcal{L}_{FDT}$. A *selection transformation* for $\leq_{\mathcal{R}}$ is a function $\Theta_{\mathcal{R}} : \mathcal{L}_{FDT} \rightarrow \mathcal{L}_{FDT}$, such that for all $I, S \in \mathcal{L}_{FDT}$:

$$I \leq_{\mathcal{R}} S \quad \text{implies} \quad I \leq_{\mathcal{R}} \Theta_{\mathcal{R}}(S)$$

□

Proposition 6.19
If the test derivation $\Pi_{\mathcal{R}}$ is sound for $\leq_{\mathcal{R}}$, and $\Theta_{\mathcal{R}}$ is a selection transformation for $\leq_{\mathcal{R}}$, then the test derivation $\Pi_{\mathcal{R}} \circ \Theta_{\mathcal{R}}$ is sound for $\leq_{\mathcal{R}}$.

□

The definition of a selection transformation only guarantees correctness, i.e. soundness, for the resulting test suite. In order to obtain test suites that are useful, transformations must be developed that take into account test selection criteria, e.g. those of the previous sections.

Apart from avoiding overproduction of test cases, performing test selection by transforming the specification has the advantage that information about the structure of the specification can be used in the selection process. A specification written as a behaviour expression has a certain structure in terms of how the specification is built from simpler behaviour expression, such as processes composed in parallel or in sequence. If it can be assumed that this structure is reflected in the structure of the implementation, it can be used to guide the test selection process. For example, if a process is composed of two independently parallel processes, it is not necessary to test all possible interleavings of actions of those processes [HGD92, VSZ92]. Such structure information is lost if we select from an (unordered) set of test cases. (cf. Specification styles in [VSSB91].)

6.5.1 Specification Selection for Labelled Transition Systems

The approach of test selection by transformation of the specification is elaborated for labelled transition systems with the implementation relation **conf**. The selection transformation is expressed by the extension relation **ext** introduced in [BSS87].

Definition 6.20 ([BSS87])
S_1 **ext** S_2 $=_{def}$ $traces(S_1) \supseteq traces(S_2)$ and S_1 **conf** S_2

□

Proposition 6.21
Any $\Theta_{\mathbf{conf}} : \mathcal{LTS} \rightarrow \mathcal{LTS}$ satisfying S **ext** $\Theta_{\mathbf{conf}}(S)$ for all $S \in \mathcal{LTS}$, is a selection transformation for **conf**.

□

The *restriction operator* on labelled transition systems, introduced in [Mil80], defines a selection transformation for **conf**. The restriction operator $\cdot\backslash A$ prunes all branches labelled with an action in $A \subseteq L$.

Definition 6.22
The *restriction operator* on labelled transition systems, $\cdot\backslash A : \mathcal{LTS} \rightarrow \mathcal{LTS}$, with $A \subseteq L$, is defined by the inference rule $I1_R$:

$$\frac{S \xrightarrow{\mu} S'}{S\backslash A \xrightarrow{\mu} S'\backslash A} \ , \ \mu \notin A \qquad (I1_R)$$

□

Proposition 6.23
For all $A \subseteq L$: S **ext** $S\backslash A$

□

Propositions 6.21 and 6.23 give a method for doing test selection by transforming a labelled transition system. Let S be the specification then we can select a set of actions L_{sel} for which we wish to derive test cases. The set with which S is restricted is the complement of L_{sel}. The transformed specification is then $S' = S\backslash(L\backslash L_{sel})$. Now according to the propositions any sound test case derived from S' is also sound for S.

Example 6.24
Consider the specification S expressed in $\mathcal{BEX}_v$:

$$S \ = \ a?x; b!x; \mathbf{stop}$$

The label set of S is $\{a, b\} \times \mathbf{N}$. We select $L_{sel} = \{a, b\} \times \{1, 10\}$:

$$\begin{array}{rcl} S' & = & (a?x; b!x; \mathbf{stop})\backslash(\ \{\{a, b\} \times (\mathbf{N}\backslash\{1, 10\}\ )\ ) \\ & = & a!1; b!1; \mathbf{stop} \ \Box \ a!10; b!10; \mathbf{stop} \end{array}$$

A **conf**-complete set of test cases for S' is

$$\begin{array}{ll} \{ & a!1; b!1; \mathbf{stop}, \\ & a!10; b!10; \mathbf{stop} \quad \} \end{array}$$

These two test cases can also be derived from S; moreover infinitely many others:

$$\begin{array}{ll} \{ & a!0; b!0; \mathbf{stop}, \\ & a!2; b!2; \mathbf{stop}, \\ & a!11; b!11; \mathbf{stop}, \\ & \vdots \qquad\qquad\qquad\quad \} \end{array}$$

□

Example 6.25
Now change S to: $S \;=\; a?x; a!(x+x); \mathbf{stop}$

With $L_{sel} = \{a\} \times \{1, 10\}$:

$$S' \;=\; a!1; \mathbf{stop} \;\Box\; a!10; \mathbf{stop}$$

with the test cases:

$$\begin{array}{ll} \{ & a!1; \mathbf{stop}, \\ & a!10; \mathbf{stop} \;\; \} \end{array}$$

These test cases are correct, but not exactly what we would like to have: the second transition is never tested.

□

The problem in example 6.25 is caused by the fact the values combined with the first action a are necessarily the same as the values of the second action a. In the restriction operator the set A of restricted actions is fixed. However, for a selection transformation it is not necessary that A is fixed; A may vary for every state of S as long as for a given σ A is the same for all $S' \in S \,\mathbf{after}\, \sigma$. This is expressed by the generalized restriction operator, which uses a function $\alpha : L^* \to \mathcal{P}(L)$ for the set of restricted actions after σ.

Definition 6.26
The *generalized restriction operator* on labelled transition systems, $\cdot\backslash\langle\alpha, \rho\rangle : \mathcal{LTS} \to \mathcal{LTS}$, with $\alpha : L^* \to \mathcal{P}(L)$, and $\rho \in L^*$, is defined by the inference rules $I2_R$ and $I3_R$:

$$\frac{S \xrightarrow{a} S'}{S\backslash\langle\alpha, \rho\rangle \xrightarrow{a} S'\backslash\langle\alpha, \rho{\cdot}a\rangle} \;,\quad \tau \neq a \notin \alpha(\rho) \qquad (I2_R)$$

$$\frac{S \xrightarrow{\tau} S'}{S\backslash\langle\alpha, \rho\rangle \xrightarrow{\tau} S'\backslash\langle\alpha, \rho\rangle} \qquad (I3_R)$$

□

Proposition 6.27
For all $\alpha : L^* \to \mathcal{P}(L)$ and $\rho \in L^*$: S **ext** $S\backslash\langle\alpha, \rho\rangle$

□

Corollary 6.28
For all $S \in \mathcal{LTS}$, sound test derivations $\Pi_{\mathbf{conf}}$ for **conf**, $\alpha : L^* \to \mathcal{P}(L)$, and $\rho \in L^*$,

$$\Pi_{\mathbf{conf}}(S\backslash\langle\alpha, \rho\rangle)$$

is a sound test suite for S with respect to **conf**.

□

Example 6.29
Again example 6.25, now with:

$$\begin{aligned} \alpha(\epsilon) &= L\backslash(\,\{a\} \times \{1, 10\}\,) \\ \alpha(\langle a, v\rangle) &= L\backslash(\,\{a\} \times \{2, 20\}\,) \quad \text{for any } v \in \mathbf{N} \end{aligned}$$

From the transformed specification

$$S' = a!1; a!2; \mathbf{stop} \,\Box\, a!10; a!20; \mathbf{stop}$$

the following test cases are derived

$$\{ \quad a!1; a!2; \mathbf{stop}, \quad a!10; a!20; \mathbf{stop} \quad \}$$

□

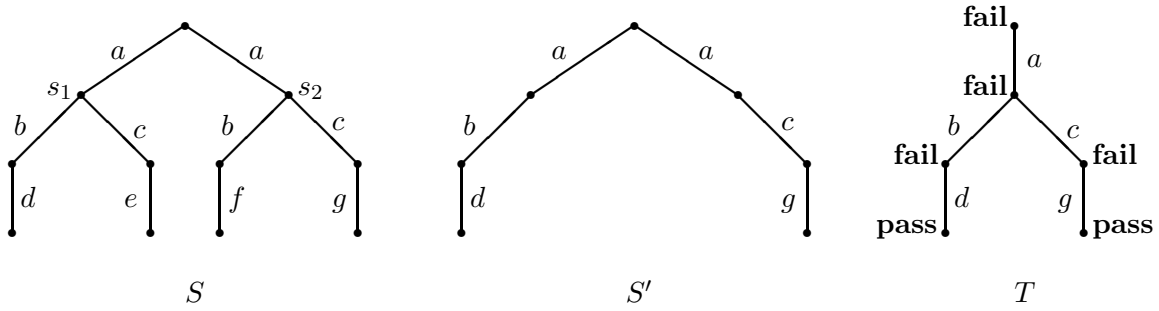


Figure 6.11: Invalid restriction.

Remark 6.30
The function $\alpha : L^* \to \mathcal{P}(L)$ in the definition of the generalized restriction operator cannot be replaced by a function on the states of the specification. Consider figure 6.11: S' is obtained from S by restricting in state s_1 to $\{b\}$, and in s_2 to $\{c\}$. But the test case T derived from S' is not sound for S.

□

6.6 Remarks on Extensions

This chapter has introduced in section 6.2 a formal framework for the study of the problems of test selection on a high level of abstraction. There are many ways in which the theory may be applied and refined. A few items are mentioned.

We used the probability that an erroneous implementation is detected, as the value of a test suite. In doing that we assumed that detection of an error, i.e. passing a test case can be expressed by the predicate **passes**. Due to nondeterminism in implementations this is not always a correct assumption: in practice the verdict **pass** gives a probability

that an implementation does not have the failure tested by the test case (cf. remark 2.2 in section 2.2.6). This implies that we have probabilities on two levels: in passing a test case, and in combining test cases into test suites. This inspires to study the problems of test selection in a fuzzy logic for test purposes. This could also provide a link with the work on probabilistic testing in [LS89, Chr90b].

For realistic applications the problem of test selection must be studied in the context of an infinite number of test cases. Many of the definitions and solutions in this paper can be adapted to cover the infinite case, but the resulting characterizations of valuations are not effective. An obvious approach to this problem would seem to define a proper notion of approximation for valuations. Also an infinite set of test purposes could be considered, more specificly the full set of requirements S (sections 2.2.1 and 2.2.3). A possible approach is to give a finite partioning of S. Each partition class defines conjunction of test purposes. This has strong resemblance to a *test group objective* of ISO9646 [ISO91a, part 1, section 3.6.6, and part 2, section 10.3].

Value and cost of a test suite were not related, implying that test selection is an optimization problem over two variables. They can be related if the value is also expressed as a cost, viz. the negative costs caused by a malfunctioning implementation in terms of repair and damage [BDD$^+$92]. The optimization problem can then be reformulated as follows: select a test suite such that the cost of executing that test suite is equal to the cost of expected repair and damage caused by failures not detected by that test suite.

To obtain an optimal test suite by using the technique of specification selection, valuations and costs must be related to the specification. If we could transform a valuation of test suites to a valuation of (parts or aspects of) behaviour expressions, it would be possible to use the optimization techniques of section 6.2 with the specification selection technique of section 6.5.

Chapter 7

Concluding Remarks

7.1 Conclusion

We have studied a formal approach to the problems of conformance testing. This has been done independently from any specific specification formalism, as well as applied to the specification formalism of labelled transition systems and formal languages based on labelled transition systems.

We started our formal approach in chapter 2 by giving a formal interpretation to concepts used in the current practice of protocol conformance testing, as reflected in the standard ISO IS-9646 'Conformance Testing Methodology and Framework' [ISO91a]. Chapter 2 did not present new theories, but it related existing, informal approaches in conformance testing to existing formal theories in the realm of processes and concurrency.

The formal interpretation of the meaning of conformance in ISO9646 corresponds to a logical specification of protocols, where specifications consist of sets of formulae, and conformance corresponds to satisfaction of these formulae. It was shown that this notion of conformance is closely related to the notion of conformance as a preorder relation on a behavioural specification formalism, which is common in the theory of concurrency. Such a relation was called an implementation relation. Next the test generation process of ISO9646 has been discussed and formally interpreted. The derivation of test purposes from conformance requirements corresponds to the selection of a finite set of conformance requirements. A generic test case is a process that describes an ideal environment that can decide about satisfaction of a test purpose. An abstract test case describes a test that takes into account the test context in which the implementation is tested. A test case is valid if it really tests the requirement for which it was developed.

Having developed a formal framework, the chapters 3, 4, 5, and 6 contribute to filling it in with specific techniques, most of them based on the formalism of labelled transition systems.

Chapter 3 elaborated on equivalence and implementation relations for labelled transition systems in the case that there is no test context. The relations testing equivalence $\approx_{te}$, testing preorder $\leq_{te}$, and conformance **conf** were discussed from the point of view of observations: two processes are related if the observations that can be made using all possible tests, can be related. To do this test cases and observations were formalized. The relation **conf** was selected as a reasonable candidate to express correctness of implementations with respect to specifications for the purpose of conformance testing with synchronous communication. Different kinds of test cases and verdict assignments were defined for it. Logical concepts were studied for **conf**-theories, where it turned out that a logically independent **conf**-theory cannot always be found. In the development of test derivation algorithms this problem was encountered in another form, preventing the unique optimization of test suites.

In chapter 4 we studied algorithms that derive **conf**-test suites systematically from a labelled transition system specification S. Also compositional test derivation from simple languages with labelled transition system semantics was studied. A main issue was representation of infinite sets of labels in a finite manner to obtain implementable algorithms. Although the languages are rather simple they have many features also appearing in more complicated languages, e.g. LOTOS [BB87, ISO89b].

In chapter 5 a test context was introduced, and its influence on conformance testing was investigated. The difference between testers that communicate synchronously with the implementation, and ones that communicate asynchronously, has been studied within the realm of the specification formalism of labelled transition systems. To describe asynchronous communication in terms of synchronous communication a queue operator on labelled transition systems was introduced. This queue operator models a pair of queues via which a labelled transition system communicates with its environment.

The theory of testing equivalence has been elaborated for this queue model. Queue equivalence $\approx_{\mathcal{Q}}$ has been defined as testing equivalence of processes in a queue context. It is the smallest equivalence that can be distinguished by other processes when communicating via queues. It was shown that queue equivalence can be characterized by two sets of traces of actions: the traces that a queue context can perform, and the deadlock traces, which are the traces that may end in deadlock. Compared to synchronous testing equivalence, where failure pairs, i.e. pairs of traces and sets of actions are needed, this looks simpler. The traces of a queue context are indeed easily characterized and related to the original specification, but the deadlock traces exhibit more complex characteristics. They cannot easily be related to traces of the original specification.

Using the queue model to formalize asynchronous communication it was shown that the traditional synchronous testing theory for labelled transition systems outlined in chapters 3 and 4 is not applicable when testing is asynchronous:

- the implementation relations used for synchronous testing are not testable in an asynchronous context;
- test cases derived from specifications to be used for synchronous testing reject

correct implementations when testing asynchronously;

- implementations that have been tested synchronously according to **conf** may behave incorrectly in an asynchronous environment.

This means that the implementation relation, i.e. the notion of conformance, that can be determined by testing, and the validity of test cases crucially depend on the test method that is used for conformance testing.

We have proposed different implementation relations for asynchronous testing. The relation $\leq_{\mathcal{O}}$ has interesting theoretical properties, whereas for conformance testing **aconf** and **asco** look more convenient. It was shown how to derive test cases for a class of these relations. The transformation of the derived test cases to TTCN, the standardized test notation [ISO91a, part 3], was briefly discussed. This transformation is possible for asynchronous test cases as opposed to the synchronous test cases derived in chapter 4 due to the inherent asynchronous nature of TTCN.

Chapter 6 studied the problems of test selection. Using algorithms for test derivation, e.g. those developed in chapters 4 and 5, usually too many test cases can be derived, Execution of all derived test cases is unfeasible or too expensive. A framework for test selection was introduced, which is very general, and only relies on the existence of a predicate that asserts the (un)successful application of a test to an implementation. The concept of valuation was introduced to represent the informal knowledge such as experience and the use of heuristics, that in practice plays an important role in the selection of tests. We have shown three methods for defining valuations, viz.

- assigning weights to the rejection of specific error cases,
- assigning weights to the equivalence classes, e.g. classes of $\approx_{\mathcal{K}}$, and
- assigning weights to combinations of test purposes/failures.

It was shown that a valuation based on a stochastic error model fits naturally within the framework. A strong point of the framework is that it allows for an easy combination of selection criteria based on the importance of failures and those based on the probability of occurrence of failures. Finally, a technique to do test selection by means of a transformation of the specification has been described and applied to labelled transition systems with the implementation relation **conf**. This technique avoids the overproduction caused by first generating too many test cases, and then selecting from them.

7.2 Open Problems

In the previous chapters a few open problems and suggestions for further research were already discussed. In this section some other items are mentioned.

The formal framework The framework as presented in chapter 2, is by no means complete, and some open questions remain. At some points simplifications and choices

for formalizations were made. These choices are open for further discussion, depending on investigations whether the presented formalizations are workable ones. An example is the relation between physical objects (the IUT) and formal objects (the specification, requirements). We assumed that implementations and test application can be formally modelled, and that observations calculated for the class of models are also valid for the physical implementations. A more elaborate identification of the assumptions underlying this approach is needed. Also the nature of the PIXIT as an interpretation function between formal objects and concrete objects (section 2.2.9) needs further study.

For validation of the usefulness of the framework it should also be applied to other specification formalisms. A possibility is to investigate how finite state machine (FSM) based formal test generation methods, like Unique Input/Output sequences, transition tours, etc., fit within the presented framework [ADLU88, BU91, SL89]. Moreover, if these FSM methods can be studied in the presented framework we have a handle to compare FSM based test derivation with labelled transition system based test derivation.

Implementation relations It was discussed that it is important to decide about the language of requirements $\mathcal{L}_R$ for a particular application, or equivalently, to choose an implementation relation, since a formal behaviour specification in itself does not uniquely define the class of conforming implementations. There are many possibilities for implementation relations; only a few of them were presented in this thesis. Although implementation relations might differ for different applications, for any specific application, implementers, testers, and users of systems need to have the same notion of what constitutes a conforming implementation. At this moment there is no such agreement, which leads to omitting the definition of an implementation relation in formal specifications. For example, the formal specification of the Transport Protocol in LOTOS does not define or refer to an implementation relation, which implies that this formal specification in fact does not define what conforming implementations are [ISO92].

The question which implementation relation to use in the realm of conformance testing is even more complicated, since it was shown in chapter 5 that the notion of conformance that can be determined by testing depends on the test method or test context. The question raised then is when conformance of an implementation to a specification can be claimed. Can conformance be claimed with respect to a specification, or can it only be claimed with respect to a specification together with a specific test context? Do we require a strong, synchronous relation like $\leq_{te}$ for conformance, even when this relation cannot be tested, or is a weaker, asynchronous relation like $\leq_{\mathcal{O}}$ sufficient, knowing that in a synchronous environment an implementation tested according to $\leq_{\mathcal{O}}$ may behave non-conforming? An important aspect is the eventual environment in which an implementation will be used, i.e. whether the context is part of the system being tested, or whether the context is only present during testing (cf. section 2.2.6). In the second case we are interested whether I conforms to S, while in the first case we are more interested in conformance of $\mathcal{C}[I]$ with respect to $\mathcal{C}[S]$.

Test derivation tools The algorithms for test derivation presented in chapters 4 and 5, turn out to be so complex, that they cannot be applied to any realistically sized specification without the help of software tools. Hence, an important item for future research is the implementation of the algorithms in tools.

Test case transformation based on contexts Synchronous test cases were derived from the specification, while asynchronous test cases were derived from the combination of specification and queue context. A topic of interest is to see whether test cases can be transformed into each other, especially whether asynchronous test cases can be obtained from the synchronous ones. More generally, transformations from generic into abstract test cases can be investigated, based on a formal description of the test context. If such transformations exist for certain contexts, we can use them to derive test cases. Otherwise test cases for testing in context (abstract test cases) must be derived directly from the composition of specification and test context, and this derivation must be repeated for each new context.

Design for testability In Integrated Circuit design it is usual to consider the testability already in the design phase. Part of the functionality of an IC is especially designed for the purpose of testing the chip when it has been realized. In some (natural language) protocol specifications an analogous approach is taken: the specification prescribes certain testing functions to be implemented. Usually these consist of making the internal state of the implementation visible to the tester. It can be investigated whether this approach can also be used for conformance testing based on formal specifications, what the consequences are for the formal specifications, and whether this could lead to a 'test-oriented' specification style (cf. [VSSB91]).

Testing and verification Verification of properties is mathematically complete and formally correct, however, most of the current verification methods are only applicable to small problems. Testing of properties is usually not complete, but it is applicable to large problems. An approach can be investigated where testing and verification are combined. Some properties of an implementation are tested, while others are verified; test results can occur as steps in verification proofs, and testing can be performed on the basis of proved properties.

Application Finally, the presented theory has been developed to be used in conformance testing. Its application to conformance testing of a realistic protocol implementation should show its usefulness and shortcomings.

Appendix A

Mathematical Preliminaries

This appendix describes some mathematical concepts and introduces some notations. The ingredients are some notations for sets, strings and traces, relations, equivalences, partitions, orders, posets, and well-foundedness.

Notation for sets A set is a collection of elements. It can be denoted by enumerating its elements: $\{a, b, c, \ldots\}$, or by stating a property P that the elements satisfy: $\{x \mid P(x)\}$. The empty set is denoted by $\emptyset$, $x \in V$ expresses that x is an element of V, and $x \notin V$ expresses that x is not. In general, / through a symbol denotes its negation. $\subseteq$ denotes the subset relation, including equality, while $\subset$ is a strict subset, so not including equality. Analogously for $\supseteq$ and $\supset$. Union $\cup$ and intersection $\cap$ are also applicable to sets of sets: $\bigcup V$ is the union of all elements in V (which must be a set of sets). The difference of the sets V and W is denoted by $V \backslash W$.

Two particular sets that are used, are $\mathbf{N}$: the set of natural numbers, and $\mathbf{R}_{\geq 0}$: the set of non-negative real numbers.

Power set The power set of a set V, $\mathcal{P}(V)$, is the set consisting of all subsets of V.

Cartesian product Let V and W be sets, then the Cartesian product of V and W is the set of all ordered pairs $\langle x, y \rangle$, such that $x \in V$ and $y \in W$: $V \times W = \{\langle x, y \rangle \mid x \in V, y \in W\}$. Analogously the generalized Cartesian product $V_1 \times V_2 \times \ldots \times V_n$ is the set of ordered n-tuples $\langle x_1, x_2, \ldots, x_n \rangle$ such that $x_i \in V_i$ for $1 \leq i \leq n$. The notation V^n denotes the set of n-tuples in the generalized Cartesian product of V with itself.

A *string*, *sequence*, or *trace* is an element of V^n for some n; V^* is the set of all strings over V:

$$V^* \quad =_{def} \quad \bigcup \{V^n \mid n = 0, 1, 2, 3, \ldots\}$$

The only string in V^0 is the empty string, denoted by ϵ. Moreover, we define:

- *length*: if a string $\sigma \in V^n$ then n is the length of σ, denoted by $|\sigma|$; note that strings in V^* have finite length.
- *concatenation*: if $\sigma_1, \sigma_2 \in V^*$, then $\sigma_1\sigma_2$ is the string in V^* that is the concatenation of σ_1 and σ_2; it is obtained by putting σ_2 at the back end of σ_1.
- *restriction*: let $W \subseteq V$, $\sigma \in V^*$, then $\sigma\lceil W$ is the restriction of σ to the elements of W:
  - $\epsilon\lceil W \;=_{def}\; \epsilon$
  - $(x{\cdot}\sigma)\lceil W \;=_{def}\; \begin{cases} x{\cdot}(\sigma\lceil W) & \text{if } x \in W \\ \sigma\lceil W & \text{if } x \notin W \end{cases}$

Relation Let V and W be sets, then a relation $\mathcal{R}$ between V and W is a subset of their Cartesian product: $\mathcal{R} \subseteq V \times W$. The element $x \in V$ is related to $y \in W$ by $\mathcal{R}$ if $\langle x, y\rangle \in \mathcal{R}$; we also write $x\mathcal{R}y$. If $\langle x, y\rangle \notin \mathcal{R}$ we write $x \not\mathcal{R} y$. Analogously, an n-ary relation is a subset of $V_1 \times V_2 \times \ldots \times V_n$.

The *inverse relation* of $\mathcal{R}$ is $\mathcal{R}^{-1}$: $\langle x, y\rangle \in \mathcal{R}$ iff $\langle y, x\rangle \in \mathcal{R}^{-1}$.

A *function* f from set V to set W, $f : V \to W$, is a relation with the property that every $a \in V$ relates to exactly one $b \in W$. This unique b is written as $b = f(a)$. A *bijection* is a function where every $b \in W$ is related to exactly one $a \in V$.

Since relations are sets, the set operations can be applied: if $\mathcal{R}_1, \mathcal{R}_2 \subseteq V \times W$, then $\mathcal{R}_1 \cap \mathcal{R}_2$ is the relation on V defined by

$$\langle x, y\rangle \in \mathcal{R}_1 \cap \mathcal{R}_2 \;=_{def}\; \langle x, y\rangle \in \mathcal{R}_1 \text{ and } \langle x, y\rangle \in \mathcal{R}_2$$

A relation $\mathcal{R}$ *on* V is a subset of $V \times V$. For such relations some important properties are defined:

- *reflexivity*: $\forall x \in V : x\mathcal{R}x$
- *transitivity*: $\forall x, y, z \in V : x\mathcal{R}y$ and $y\mathcal{R}z$ imply $x\mathcal{R}z$
- *symmetry*: $\forall x, y \in V : x\mathcal{R}y$ implies $y\mathcal{R}x$
- *anti-symmetry*: $\forall x, y \in V : x\mathcal{R}y$ and $y\mathcal{R}x$ imply $x = y$
- *linearity*: $\forall x, y \in V : x\mathcal{R}y$ or $y\mathcal{R}x$

Equivalence An equivalence on V is a relation that is reflexive, transitive and symmetric.

For an equivalence relation $\mathcal{E}$ on V the *equivalence class* of an element $a \in V$, $[a]_\mathcal{E}$, is the set of all elements in V that are equivalent to a:

$$[a]_\mathcal{E} \;=_{def}\; \{b \in V \mid a\mathcal{E}b\}$$

The set of all such classes is the *quotient* of V with respect to $\mathcal{E}$:

$$V/\mathcal{E} \;=_{def}\; \{[a]_\mathcal{E} \mid a \in V\}$$

The quotient $V/\mathcal{E}$ forms a *partition* of V. A partition is a division of V in nonempty subsets of V such that these subsets do not overlap, and the union of all subsets is equal to V: $\Xi = \{\xi_1, \ldots, \xi_n\}$ is a partition of V, if for all $1 \leq i \leq n$: $\emptyset \neq \xi_i \subseteq V$, $\bigcup \Xi = V$, and $\xi_i \cap \xi_j \neq \emptyset$ implies $i = j$.

Order An order on V is a relation that expresses the intuition of ordering the elements of V, so that we can talk about elements being 'smaller' or 'larger' than other elements. We denote an order relation by $\trianglelefteq$, its inverse by $\trianglerighteq$. Different kinds of orders can be defined depending on their properties.

A *preorder* is reflexive and transitive; a *partial order* is an anti-symmetric preorder; a *linear* or *total order* is a linear partial order. For these orders, because of reflexivity, always $x \trianglelefteq x$; in a *strict order* this is not the case: $\triangleleft$ is the strict order corresponding to $\trianglelefteq$: $x \triangleleft y \;\; =_{def} \;\; x \trianglelefteq y$ and $x \neq y$.

A set together with a partial order on it is called a *poset*: $\langle V, \trianglelefteq \rangle$.

Once we have 'smaller' and 'larger' elements we can define *minimal elements* of a poset, i.e. elements without smaller elements:

$$\text{let } W \subseteq V \text{ then } m \text{ is a } \textit{minimal element} \text{ of } W \text{ if } m \in W \text{ and } \forall x \in W : x \not\trianglelefteq m$$

Since uniqueness and existence of minimal elements is not guaranteed, it makes sense to define a set of minimal elements:

$$min_{\trianglelefteq}(W) \;\; =_{def} \;\; \{m \in W \mid m \text{ is minimal element of } W\}$$

The existence of minimal elements is important: the poset $\langle V, \trianglelefteq \rangle$ is *well-founded* if all nonempty subsets of V have a minimal element.

Well-foundedness means that there is no infinite sequence of elements that are strict smaller, i.e. no sequence of different x_i, such that

$$\ldots \triangleleft x_3 \triangleleft x_2 \triangleleft x_1 \triangleleft x_0$$

Closure Let $\langle V, \trianglelefteq \rangle$ be a poset, then the *right-closure* of a set $W \subseteq V$, $rcl_{\trianglelefteq}(W)$ or $\overline{W}$, is that set together with all elements that are larger:

$$\overline{W} \;\; =_{def} \;\; \{y \in V \mid \exists x \in W : x \trianglelefteq y\}$$

A set is *right-closed* if it is equal to its right-closure.

If a set is extended with all elements that are larger than the elements of that set, then the minimal elements are not affected:

Proposition A.1
Let $\langle V, \trianglelefteq \rangle$ be a poset, then for $A \subseteq V$: $\quad min_{\trianglelefteq}(A) = min_{\trianglelefteq}(\overline{A})$.

□

Proof (proposition A.1)

$\subseteq$: Let $m \in min_{\trianglelefteq}(A)$ then $m \in A$ and $\forall x \in A : x \trianglelefteq m$ implies $x = m$.
We have to prove: $m \in \overline{A}$ and $\forall y \in \overline{A} : y \trianglelefteq m$ implies $y = m$:

- $m \in A$ implies $m \in \overline{A}$;
- let $y \in \overline{A}$ with $y \trianglelefteq m$, then $\exists y' \in A : y' \trianglelefteq y$;
  this implies $y' \trianglelefteq y \trianglelefteq m$, hence $y' = m$ and $y = m$.

$\supseteq$: Let $m \in min_{\trianglelefteq}(\overline{A})$ then $m \in \overline{A}$ and $\forall x \in \overline{A} : x \trianglelefteq m$ implies $x = m$.
We have to prove: $m \in A$ and $\forall y \in A : y \trianglelefteq m$ implies $y = m$:

- $m \in \overline{A}$ iff $\exists m' \in A : m' \trianglelefteq m$; since $m' \in \overline{A} : m' = m$, so $m \in A$;
- since $A \subseteq \overline{A} : \forall y \in A : y \trianglelefteq m$ implies $y = m$.

□

Proposition A.1 can be used to represent a right-closed set by the set of its minimal elements. However, minimal elements need not exist. Well-foundedness or finiteness guarantees existence of minimal elements.

Proposition A.2
Let $\langle V, \trianglelefteq \rangle$ be a poset, and $A_1, A_2 \subseteq V$, then $\overline{A_1} = \overline{A_2}$ iff $min_{\trianglelefteq}(A_1) = min_{\trianglelefteq}(A_2)$, if

1. $\langle V, \trianglelefteq \rangle$ is well-founded, or
2. A_1, A_2 are finite.

□

Proof (proposition A.2)

only if: $\overline{A_1} = \overline{A_2}$
implies $min_{\trianglelefteq}(\overline{A_1}) = min_{\trianglelefteq}(\overline{A_2})$ (∗ proposition A.1 ∗)
implies $min_{\trianglelefteq}(A_1) = min_{\trianglelefteq}(A_2)$

if: Let $x_1 \in \overline{A_1}$, then we have to prove $x_1 \in \overline{A_2}$.
$x_1 \in \overline{A_1}$ implies $\exists x_0 \in A_1 : x_0 \trianglelefteq x_1$.
Define $X_0 = \{x \in A_1 \mid x \trianglelefteq x_0\}$; $\emptyset \neq X_0 \subseteq A_1 \subseteq V$.
There exists a minimal element $m_0 \in X_0$, because

either: $\langle V, \trianglelefteq \rangle$ is well-founded, hence every nonempty subset of V has a minimal element,

or: A_1 is finite, so X_0 is finite, and every nonempty finite set has a minimal element.

This m_0 is also a minimal element of A_1, because

both: $m_0 \in A_1$,

and: $\forall y \in A_1 : y \trianglelefteq m_0$ implies $y = m_0$:
suppose $y \trianglelefteq m_0$; with $m_0 \trianglelefteq x_0$ (since $m_0 \in X_0$) and transitivity: $y \trianglelefteq x_0$, hence $y \in X_0$;
combining $y \trianglelefteq m_0$, $y \in X_0$ with m_0 is the minimal element of X_0: $y = m_0$.

So $m_0 \in min_{\trianglelefteq}(A_1) = min_{\trianglelefteq}(A_2)$, and since $m_0 \trianglelefteq x_0 \trianglelefteq x_1$, using right-closedness of $\overline{A_2} : x_1 \in \overline{A_2}$.
Symmetrically $\overline{A_2} \subseteq \overline{A_1}$ is proved.

□

Analogous definitions and propositions can be given for the dual concepts *maximal element*, $max_{\trianglelefteq}(W)$, *co-well-foundedness*, *left-closure* $lcl_{\trianglelefteq}$, and *left-closedness*.

Appendix B

Proofs

B.1 Chapter 1 (Introduction)

Proposition 1.10

1. $\langle L^*, \preceq \rangle$ is a well-founded poset.
2. $\sigma_1 \preceq \sigma_2$ and $\sigma_2 \in \mathit{traces}(S)$ imply $\sigma_1 \in \mathit{traces}(S)$
3. $\mathit{traces}(S)$ is *prefix-closed*, i.e. it is left-closed with respect to $\preceq$
4. $\mathit{min}_{\preceq}(\, \mathit{traces}(S)\,) = \{\epsilon\}$

□

Proof (proposition 1.10)

1. $\preceq$ is a relation on L^* that is reflexive, transitive, and anti-symmetric:

   reflexivity: According to definition 1.9.2: take $\sigma' = \epsilon$.

   transitivity: Suppose $\sigma_1 \preceq \sigma_2$ and $\sigma_2 \preceq \sigma_3$, with $\sigma_1 \cdot \sigma_1' = \sigma_2$ and $\sigma_2 \cdot \sigma_2' = \sigma_3$. Then $\sigma_1 \cdot \sigma_1' \cdot \sigma_2' = \sigma_3$, thus $\sigma_1 \preceq \sigma_3$.

   anti-symmetry: Suppose $\sigma_1 \preceq \sigma_2$ and $\sigma_2 \preceq \sigma_1$, with $\sigma_1 \cdot \sigma_1' = \sigma_2$ and $\sigma_2 \cdot \sigma_2' = \sigma_1$. Then $\sigma_1 \cdot \sigma_1' \cdot \sigma_2' = \sigma_1$, thus $\sigma_1' = \sigma_2' = \epsilon$, and $\sigma_1 = \sigma_2$.

   Moreover, $\preceq$ is well-founded:

   $$\forall T : \emptyset \neq T \subseteq L^* : \exists \sigma_m \in T : \forall \sigma \in T : \sigma \not\prec \sigma_m$$

   Assume non-well-foundedness: $\exists T_0 : \emptyset \neq T_0 \subseteq L^* : \forall \sigma_m \in T_0 : \exists \rho_m \in T_0 : \rho_m \prec \sigma_m$.
   This means that, since $T_0 \neq \emptyset$, there is $\sigma_0 \in T_0$ with: $\exists \rho_1 \in T_0 : \rho_1 \prec \sigma_0$.
   Again for ρ_1: $\exists \rho_2 \in T_0 : \rho_2 \prec \rho_1$, and so on. Hence there exists an infinite sequence of different traces

   $$\ldots \prec \rho_4 \prec \rho_3 \prec \rho_2 \prec \rho_1 \prec \sigma_0$$

   However, $\sigma_1 \prec \sigma_2$ implies $|\sigma_1| < |\sigma_2|$, and since $|\sigma_0| \in \mathbf{N}$, this would imply the existence of an infinite sequence of decreasing natural numbers. Such a sequence

does not exist (well-foundedness of $\mathbf{N}$), hence the sequence of traces does not exist, hence the assumption of non-well-foundedness is not correct.

2. Since $\sigma_1 \preceq \sigma_2$ implies $\exists \sigma' : \sigma_2 = \sigma_1 \cdot \sigma'$:

$$\begin{array}{ll} & \sigma_2 \in \mathit{traces}(S) \\ \text{implies} & S \stackrel{\sigma_1 \cdot \sigma'}{\Longrightarrow} \\ \text{implies} & S \stackrel{\sigma_1}{\Longrightarrow} \\ \text{implies} & \sigma_1 \in \mathit{traces}(S) \end{array}$$

3. To prove: $\mathit{traces}(S) = \mathit{lcl}_{\preceq}(\mathit{traces}(S))$:

   $\subseteq$: From the definition of $\mathit{lcl}_{\preceq}$ using reflexivity of $\preceq$.

   $\supseteq$: Let $\sigma \in \mathit{lcl}_{\preceq}(\mathit{traces}(S))$, then there is $\sigma_m \in \mathit{traces}(S)$ such that $\sigma \preceq \sigma_m$. Using proposition 1.10.2: $\sigma \in \mathit{traces}(S)$.

4. ϵ is minimal: $\epsilon \in \mathit{traces}(S)$ ($S \stackrel{\epsilon}{\Longrightarrow}$ for any S), and $\forall \sigma \in \mathit{traces}(S) : \sigma \not\preceq \epsilon$.
   ϵ is the only minimal element: let σ_m be minimal element too, then:
   $\forall \sigma \in \mathit{traces}(S) : \sigma \preceq \sigma_m$ implies $\sigma = \sigma_m$. Since $\epsilon \preceq \sigma_m$ for any σ_m: $\sigma_m = \epsilon$.

□

Proposition 1.14

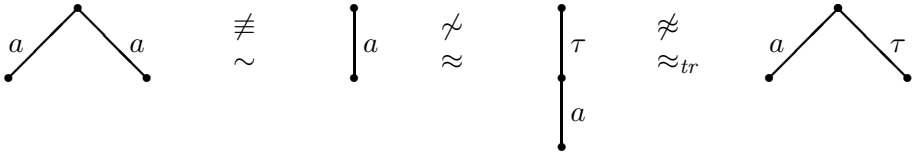
1. $\equiv$, $\sim$, $\approx$, and $\approx_{tr}$ are equivalences.
2. $\equiv \subset \sim \subset \approx \subset \approx_{tr}$

□

Proof (proposition 1.14)

1. Directly from their definitions.
2. Directly from the facts that:
   - the bijection f of 1.13.1 fulfils the requirements for $\mathcal{R}$ in 1.13.2;
   - the relation $\mathcal{R}$ of 1.13.2. fulfils the requirements for $\mathcal{R}$ is 1.13.3, which can be proved using induction on the lengh of σ;
   - from $\langle s_{0_1}, s_{0_2} \rangle \in \mathcal{R}$ in 1.13.3 it follows that $s_{0_1} \stackrel{\sigma}{\Longrightarrow}$ implies $s_{0_1} \stackrel{\sigma}{\Longrightarrow}$.

   For inequalities:



□

Proposition 1.16

1. $\approx_{tr} \; = \; \leq_{tr} \cap \geq_{tr}$
2. $\leq_{tr}$ and $\geq_{tr}$ are preorders

□

Proof (proposition 1.16)
Directly from their definitions. □

B.2 Chapter 2 (A Formal Framework for Conformance Testing)

B.2.1 Section 2.3 (Conformance as a Relation)

Proposition 2.3
Let **conforms-to** $\subseteq \mathcal{L}_{FDT} \times \mathcal{L}_{FDT}$ be defined by (2.14):

$$B_I \textbf{ conforms-to } B_S \quad =_{def} \quad \forall r \in \mathcal{L}_R : \; B_S \textbf{ spec } r \;\text{ implies }\; B_I \textbf{ sat } r$$

then

1. **conforms-to** is *reflexive* if and only if **spec** $\subseteq$ **sat**
2. **conforms-to** is *transitive* if **spec** $\supseteq$ **sat**
3. **conforms-to** is a *preorder* if **spec** $=$ **sat**
4. **conforms-to** is an *equivalence* if **spec** $=$ **sat** and negation is expressible in $\mathcal{L}_R$, i.e. if
$$\forall r \in \mathcal{L}_R, \exists\, \overline{r} \in \mathcal{L}_R, \forall B \in \mathcal{L}_{FDT} : B \textbf{ sat } \overline{r} \quad \text{iff} \quad \text{not} \;\; B \textbf{ sat } r$$

□

Proof (proposition 2.3)

1. **conforms-to** is reflexive — iff (∗ def. reflexivity ∗)
   $\forall B \in \mathcal{L}_{FDT} : B$ **conforms-to** B — iff (∗ (2.14) ∗)
   $\forall B \in \mathcal{L}_{FDT}, \forall r \in \mathcal{L}_R : B$ **spec** r implies B **sat** r — iff (∗ def. of $\subseteq$ ∗)
   spec $\subseteq$ **sat**
2. We have to prove that for all $B_1, B_2, B_3 \in \mathcal{L}_{FDT}$
   if B_1 **conforms-to** B_2 and
   B_2 **conforms-to** B_3 and
   then B_1 **conforms-to** B_3
   To prove this, let $B_1, B_2, B_3 \in \mathcal{L}_{FDT}$, and $r \in \mathcal{L}_R$,
   with B_1 **conforms-to** B_2 and B_2 **conforms-to** B_3, then
   B_3 **spec** r implies (∗ (2.14) applied to B_2 **conforms-to** B_3 ∗)
   B_2 **sat** r implies (∗ **spec** $\supseteq$ **sat** ∗)
   B_2 **spec** r implies (∗ (2.14) applied to B_1 **conforms-to** B_2 ∗)
   B_1 **sat** r
   Thus: B_3 **spec** r implies B_1 **sat** r
   i.e. (∗ (2.14) ∗) B_1 **conforms-to** B_3
3. We have to prove that **conforms-to** is reflexive and transitive (definition of preorder, appendix A). Reflexivity follows from 2.3.1; transitivity from 2.3.2.
4. We have to prove that **conforms-to** is reflexive, transitive and symmetric (definition of equivalence). Reflexivity follows from 2.3.1; transitivity from 2.3.2.
   For symmetry we have to prove that for all $B_1, B_2 \in \mathcal{L}_{FDT}$

if B_1 **conforms-to** B_2
then B_2 **conforms-to** B_1
To prove this, let $B_1, B_2 \in \mathcal{L}_{FDT}$ with B_1 **conforms-to** B_2, and let $r \in \mathcal{L}_R$, then
B_1 **spec** r iff $(*$ def. of $\overline{r}$ $*)$
notB_1 **spec** $\overline{r}$ iff $(*$ **spec** = **sat** $*)$
notB_1 **sat** $\overline{r}$ implies $(*$ contraposition (2.14) to B_1 **conforms-to** B_2 $*)$
notB_2 **spec** $\overline{r}$ iff $(*$ def. of $\overline{r}$ $*)$
B_2 **spec** r iff $(*$ **spec** = **sat** $*)$
B_2 **sat** r
Thus: B_1 **spec** r implies B_2 **sat** r
i.e. $(*$ (2.14) $*)$ B_2 **conforms-to** B_1

□

Proposition 2.4
If $\forall B_I \in \mathcal{L}_{FDT}, \exists\, B_S \in \mathcal{L}_{FDT} : \{r \in \mathcal{L}_R \mid B_S \textbf{ spec } r\} = \{r \in \mathcal{L}_R \mid B_I \textbf{ sat } r\}$,
and $\forall B_S \in \mathcal{L}_{FDT}, \exists\, B_I \in \mathcal{L}_{FDT} : \{r \in \mathcal{L}_R \mid B_I \textbf{ sat } r\} = \{r \in \mathcal{L}_R \mid B_S \textbf{ spec } r\}$,
then

conforms-to is transitive if and only if **spec** $\supseteq$ **sat**

□

Proof (proposition 2.4)
The *if*-part follows from proposition 2.3.2; the *only if*-part is proved by contraposition:

$$\begin{array}{rl} & \exists B_0, r_0 : B_0 \textbf{ sat } r_0 \text{ and } B_0 \textbf{ sp}\not\textbf{ec } r_0 \\ \text{implies} & \exists B_1, B_2, B_3 : \; B_1 \textbf{ conforms-to } B_2 \text{ and } B_2 \textbf{ conforms-to } B_3 \\ & \qquad \text{and } B_1 \textbf{ confor}\not\textbf{ms-to } B_3 \end{array}$$

Take B_2 equal to B_0,
B_1 such that $\{r \in \mathcal{L}_R \mid B_1 \textbf{ sat } r\} = \{r \in \mathcal{L}_R \mid B_2 \textbf{ spec } r\}$, (1)
B_3 such that $\{r \in \mathcal{L}_R \mid B_3 \textbf{ spec } r\} = \{r \in \mathcal{L}_R \mid B_2 \textbf{ sat } r\}$, (2)
then B_1 **conforms-to** B_2 and B_2 **conforms-to** B_3;
using B_0 **sat** r_0 and (2) : $r_0 \in \{r \in \mathcal{L}_R \mid B_3 \textbf{ spec } r\}$,
using B_0 **sp**$\not$**ec** r_0 and (1) : $r_0 \notin \{r \in \mathcal{L}_R \mid B_1 \textbf{ sat } r\}$,
hence B_3 **spec** r_0 and B_1 **sa**$\not$**t** r_0, so B_1 **confor**$\not$**ms-to** B_3.

□

B.2.2 Proofs of Section 2.4 (Examples)

Proposition 2.6
Let **conforms-to** $\subseteq \mathcal{LTS} \times \mathcal{LTS}$ be defined by (2.14):

$$B_I \textbf{ conforms-to } B_S \quad =_{def} \quad \forall r \in \mathcal{L}_R : \; B_S \textbf{ spec } r \text{ implies } B_I \textbf{ sat } r$$

and let $\mathcal{L}_{tr}$, $\mathcal{L}_{\overline{tr}}$, **spec**, and **sat** be given in definition 2.5, then

- For $\mathcal{L}_R = \mathcal{L}_{tr}$: **conforms-to** $= \leq_{tr}$
- For $\mathcal{L}_R = \mathcal{L}_{\overline{tr}}$: **conforms-to** $= \geq_{tr}$

□

Proof (proposition 2.6)

- B_I **conforms-to** B_S
  iff $\forall r \in \mathcal{L}_{tr} : B_S$ **spec** r implies B_I **sat** r
  iff $\forall \sigma \in L^* : B_S$ **spec cannot** σ implies B_I **sat cannot** σ
  iff $\forall \sigma \in L^* : B_S \overset{\sigma}{\Longrightarrow\!\!\!\!\!\!/}$ implies $B_I \overset{\sigma}{\Longrightarrow\!\!\!\!\!\!/}$
  iff $\forall \sigma \in L^* : B_I \overset{\sigma}{\Longrightarrow}$ implies $B_S \overset{\sigma}{\Longrightarrow}$
  iff $traces(B_I) \subseteq traces(B_S)$
  iff $B_I \leq_{tr} B_S$
- B_I **conforms-to** B_S
  iff $\forall r \in \mathcal{L}_{\overline{tr}} : B_S$ **spec** r implies B_I **sat** r
  iff $\forall \sigma \in L^* : B_S$ **spec can** σ implies B_I **sat can** σ
  iff $\forall \sigma \in L^* : B_S \overset{\sigma}{\Longrightarrow}$ implies $B_I \overset{\sigma}{\Longrightarrow}$
  iff $traces(B_S) \subseteq traces(B_I)$
  iff $B_I \geq_{tr} B_S$

□

Proposition 2.7
Derivation for $\mathcal{L}_{\overline{tr}}$ as defined by (2.23), (2.24), and (2.25) is sound and complete. □

Proof (proposition 2.7)

Soundness: We have to prove (2.26) for any S and r.
Let $S \vdash r$, $r =$ **can** σ_0 , then either (2.23), (2.24), or (2.25):

(2.23): If $I \models S$, then, since **can** $\sigma_0 \in S$, also $I \models$ **can** σ_0 , thus, the right-hand side of (2.26) holds.

(2.24): This implies that $\sigma_0 = \epsilon$.
From definition 2.5 it follows that $I \models$ **can** ϵ iff $I \overset{\epsilon}{\Longrightarrow}$,
and this is true for any I, thus, the right-hand side of (2.26) holds.

(2.25): This implies that there is σ_0' such that **can** $\sigma_0 \cdot \sigma_0' \in S$.
If we have $I \models S$, then we also have $I \models$ **can** $\sigma_0 \cdot \sigma_0'$,
which means (definition 2.5) $I \overset{\sigma_0 \cdot \sigma_0'}{\Longrightarrow}$.
According to proposition 1.10.2: $I \overset{\sigma_0 \cdot \sigma_0'}{\Longrightarrow}$ implies $I \overset{\sigma_0}{\Longrightarrow}$,
which means (definition 2.5) $I \models$ **can** σ_0 , thus, the right-hand side of (2.26) holds.

Completeness: This is proven by contraposition, i.e. we have to prove that

$$\text{if } S \nvdash r \text{ then } \exists I \ (\ I \models S \text{ and } I \not\models r\ )$$

Let $r =$ **can** σ_0 ; we have to find I such that S is satisfied and **can** σ_0 is not.
Take for I the labelled transition system with only branches from the initial state

for every σ with $\mathbf{can}\ \sigma \in S$: $I = \Sigma\{\sigma;\mathbf{stop} \mid \mathbf{can}\ \sigma \in S\}$. This I has the required properties:

$I \models S$: $I \stackrel{\sigma}{\Longrightarrow}$ for all $\mathbf{can}\ \sigma \in S$, thus (definition 2.5) $I \models S$.

$I \not\models \mathbf{can}\ \sigma_0$: Suppose that $I \models \mathbf{can}\ \sigma_0$, then $I \stackrel{\sigma_0}{\Longrightarrow}$, so there must be a branch of I that makes this possible, implying that there exists σ_0' such that $I \stackrel{\sigma_0 \cdot \sigma_0'}{\Longrightarrow}$ with $\mathbf{can}\ \sigma_0 \cdot \sigma_0' \in S$.
Using (2.25) we would have $S \vdash \mathbf{can}\ \sigma_0$, which contradicts $S \not\vdash r$. Hence $I \not\models \mathbf{can}\ \sigma_0$.

□

B.3 Chapter 3 (Implementation Relations)

B.3.1 Section 3.2 (Testing Equivalence)

Proposition 3.3.1
$I_1 \approx_{te} I_2 \quad \text{iff} \quad \forall t \in \mathcal{LTS} : Obs(t, I_1) = Obs(t, I_2)$

□

Proof (proposition 3.3.1)
We prove:

$$\begin{array}{ll} & \forall t \in \mathcal{LTS} : Obs(t, I_1) \subseteq Obs(t, I_2) \ \text{ and } \ Obs'(t, I_1) \subseteq Obs'(t, I_2) \\ \text{iff} & \forall t \in \mathcal{LTS} : Obs(t, I_1) \subseteq Obs(t, I_2) \end{array} \tag{1}$$

from which the proposition follows directly.

only if: Directly.

if: Let $t \in \mathcal{LTS}$, then directly $Obs(t, I_1) \subseteq Obs(t, I_2)$.
Let $\sigma \in Obs'(t, I_1)$, then $t \,\|\, I_1 \stackrel{\sigma}{\Longrightarrow}$ implying $\exists I_1' : I_1 \stackrel{\sigma}{\Longrightarrow} I_1'$ and $\exists t' : t \stackrel{\sigma}{\Longrightarrow} t'$ (2)
let $\sigma = b_1 \cdot b_2 \cdot \ldots \cdot b_m$ and $t_\sigma = b_1; b_2; \ldots; b_m; \mathbf{stop}$,

then	$\exists I_1' \ (\ t_\sigma \,\|\, I_1 \stackrel{\sigma}{\Longrightarrow} \mathbf{stop} \,\|\, I_1' \ \text{ and } \ \forall a \in L : \mathbf{stop} \,\|\, I_1' \stackrel{a}{\not\Longrightarrow}\ )$
implies	$(*$ definition 3.1.1 $*)$
	$t_\sigma \,\|\, I_1$ **after** σ **deadlocks**
implies	$(*$ definition 3.1.2 $*)$
	$\sigma \in Obs(t_\sigma, I_1)$
implies	$(*$ premiss $*)$
	$\sigma \in Obs(t_\sigma, I_2)$
implies	$t_\sigma \,\|\, I_2$ **after** σ **deadlocks**
implies	$t_\sigma \,\|\, I_2 \stackrel{\sigma}{\Longrightarrow}$
implies	$I_2 \stackrel{\sigma}{\Longrightarrow}$
implies	$(*$ using (2) $*)$
	$t \,\|\, I_2 \stackrel{\sigma}{\Longrightarrow}$
implies	$\sigma \in Obs'(t, I_2)$

So, $\sigma \in Obs'(t, I_1)$ implies $\sigma \in Obs'(t, I_2)$, which completes the proof of (1).

□

Lemma B.1
Let $t, I \in \mathcal{LTS}$, and $t_{[\sigma,A]} \in \mathcal{LT}_M$, then (using definition 3.4):

1. $\sigma \in Obs(t, I)$ iff $\exists t' \ (\ t \stackrel{\sigma}{\Longrightarrow} t' \ \text{ and } \ I$ **after** σ **refuses** $out(t')\ )$
2. $\sigma \in Obs(t_{[\sigma,A]}, I)$ iff I **after** σ **refuses** A

□

Proof (lemma B.1)

1. $\sigma \in Obs(t, I)$
   iff $(*$ definition 3.1.2 $*)$
   $t \,\|\, I$ **after** σ **deadlocks**
   iff $(*$ definition 3.1.1 $*)$
   $\exists t', I' \; (\; t \,\|\, I \stackrel{\sigma}{\Longrightarrow} t' \,\|\, I' \;$ and $\; \forall a \in L : I' \,\|\, t' \stackrel{a}{\Longrightarrow\!\!\!\!\!/} \;)$
   iff $(*$ definitions 1.7 and 1.11.1 $*)$
   $\exists t', I' \; (\; t \stackrel{\sigma}{\Longrightarrow} t' \;$ and $\; I \stackrel{\sigma}{\Longrightarrow} I' \;$ and $\; \forall a \in out(t') : I' \stackrel{a}{\Longrightarrow\!\!\!\!\!/} \;)$
   iff $(*$ definition 3.4 $*)$
   $\exists t' \; (\; t \stackrel{\sigma}{\Longrightarrow} t' \;$ and $\;$ I **after** σ **refuses** $out(t') \;)$
2. Applying lemma B.1.1, using definition 3.2:

$$t_{[\sigma,A]} \stackrel{\sigma}{\Longrightarrow} \Sigma\{a; \mathbf{stop} \mid a \in A\} \quad \text{and} \quad out(\Sigma\{a; \mathbf{stop} \mid a \in A\}) = A$$

□

Proposition 3.3.2
$I_1 \approx_{te} I_2$ iff $\forall t \in \mathcal{LT}_M : Obs(t, I_1) = Obs(t, I_2)$

□

Proof (proposition 3.3.2)
We prove:

$$\forall t \in \mathcal{LTS} : Obs(t, I_1) \subseteq Obs(t, I_2) \quad \text{iff} \quad \forall t \in \mathcal{LT}_M : Obs(t, I_1) \subseteq Obs(t, I_2) \qquad (1)$$

from which the proposition follows directly, using proposition 3.3.1.

only if: Directly from $\mathcal{LT}_M \subseteq \mathcal{LTS}$.

if: Let $t \in \mathcal{LTS}$,
then $\sigma \in Obs(t, I_1)$
implies $(*$ lemma B.1.1 $*)$
$\exists t' \; (\; t \stackrel{\sigma}{\Longrightarrow} t' \;$ and $\;$ I_1 **after** σ **refuses** $out(t') \;)$
implies $(*$ lemma B.1.2 $*)$
$\exists t' \; (\; t \stackrel{\sigma}{\Longrightarrow} t' \;$ and $\; \sigma \in Obs(t_{[\sigma,out(t')]}, I_1) \;)$
implies $(*$ premiss $*)$
$\exists t' \; (\; t \stackrel{\sigma}{\Longrightarrow} t' \;$ and $\; \sigma \in Obs(t_{[\sigma,out(t')]}, I_2) \;)$
implies $\exists t' \; (\; t \stackrel{\sigma}{\Longrightarrow} t' \;$ and $\;$ I_2 **after** σ **refuses** $out(t') \;)$
implies $\sigma \in Obs(t, I_2)$

□

Proposition 3.5

1. I **after** σ **refuses** A iff not $(\; I$ **after** σ **must** $A \;)$
2. if I **after** σ **refuses** A_1 and $A_1 \supseteq A_2$ then I **after** σ **refuses** A_2
3. if I **after** σ **must** A_1 and $A_1 \subseteq A_2$ then I **after** σ **must** A_2
4. I **after** σ **deadlocks** iff I **after** σ **refuses** L

5. I **after** σ **refuses** $\emptyset$ iff $\sigma \in traces(I)$
6. I **after** σ **refuses** A iff $\sigma \in Obs(t_{[\sigma,A]}, I)$

□

Proof (proposition 3.5)

1. not (I **after** σ **must** A)
   iff not ($\forall I' \in I$ **after** $\sigma : \exists a \in A : I' \stackrel{a}{\Longrightarrow}$)
   iff $\exists I' \in I$ **after** σ : not ($\exists a \in A : I' \stackrel{a}{\Longrightarrow}$)
   iff $\exists I' \in I$ **after** $\sigma : \forall a \in A : I' \stackrel{a}{\not\Longrightarrow}$
   iff I **after** σ **refuses** A
2. I **after** σ **refuses** A_1
   iff $\exists I' \in I$ **after** $\sigma : \forall a \in A_1 : I' \stackrel{a}{\not\Longrightarrow}$
   implies $\exists I' \in I$ **after** $\sigma : \forall a \in A_2 : I' \stackrel{a}{\not\Longrightarrow}$
   iff I **after** σ **refuses** A_2
3. I **after** σ **must** A_1
   iff $\forall I' \in I$ **after** $\sigma : \exists a \in A_1 : I' \stackrel{a}{\Longrightarrow}$
   implies $\forall I' \in I$ **after** $\sigma : \exists a \in A_2 : I' \stackrel{a}{\Longrightarrow}$
   iff I **after** σ **must** A_2
4. Application of definitions 3.1.1 and 3.4.
5. I **after** σ **refuses** $\emptyset$
   iff $\exists I' \in I$ **after** $\sigma : \forall a \in \emptyset : I' \stackrel{a}{\not\Longrightarrow}$
   iff $\exists I' \in I$ **after** σ
   iff $\sigma \in traces(I)$
6. Lemma B.1.2.

□

Theorem 3.6

$I_1 \approx_{te} I_2$ iff $\forall \sigma \in L^*, \forall A \subseteq L :$ I_1 **after** σ **must** A iff I_2 **after** σ **must** A

□

Proof (theorem 3.6)

only if: Let $\sigma \in L^*$, $A \subseteq L$, then, using proposition 3.5.1:
   I_1 **after** σ **refuses** A
   iff ($*$ proposition 3.5.6 $*$)
   $\sigma \in Obs(t_{[\sigma,A]}, I_1)$
   iff ($*$ proposition 3.3.2 $*$)
   $\sigma \in Obs(t_{[\sigma,A]}, I_2)$
   iff I_2 **after** σ **refuses** A

if: Using proposition 3.3.1, let $t \in \mathcal{LTS}$, then

$\sigma \in Obs(t, I_1)$

iff $(*$ lemma B.1.1 $*)$

$\exists t' \; (\; t \stackrel{\sigma}{\Longrightarrow} t' \;$ and $\; I_1$ **after** σ **refuses** $out(t') \;)$

iff $(*$ premiss $*)$

$\exists t' \; (\; t \stackrel{\sigma}{\Longrightarrow} t' \;$ and $\; I_2$ **after** σ **refuses** $out(t') \;)$

iff $\sigma \in Obs(t, I_2)$

□

Proposition 3.7

$\equiv \; \subset \; \sim \; \subset \; \approx \; \subset \; \approx_{te} \; \subset \; \approx_{tr}$

□

Proof (proposition 3.7)

We prove $\approx \subseteq \approx_{te}$, $\approx_{te} \subseteq \approx_{tr}$, $\approx \neq \approx_{te}$, and $\approx_{te} \neq \approx_{tr}$. The rest was proved in proposition 1.14.2.

$\approx \subseteq \approx_{te}$: Let $B_1 \approx B_2$, then there exists $\mathcal{R} \subseteq der(B_1) \times der(B_2)$ satisfying the requirements of definition 1.13.3, with $\langle B_1, B_2 \rangle \in \mathcal{R}$.

Let $\sigma \in L^*$, $A \subseteq L$, then

B_1 **after** σ **refuses** A

implies $\exists B_1' \; (\; B_1 \stackrel{\sigma}{\Longrightarrow} B_1' \;$ and $\; \forall a \in A : B_1' \stackrel{a}{\not\Longrightarrow} \;)$

implies $(* \; \langle B_1, B_2 \rangle \in \mathcal{R} \; *)$

$\exists B_2' : B_2 \stackrel{\sigma}{\Longrightarrow} B_2' \;$ and $\; \langle B_1', B_2' \rangle \in \mathcal{R}$

implies $(* \; \langle B_1', B_2' \rangle \in \mathcal{R} \;$ implies $\; \forall a \in A : B_1' \stackrel{a}{\Longrightarrow} \;$ iff $\; B_2' \stackrel{a}{\Longrightarrow} \; *)$

$\exists B_2' : \forall a \in A : B_2' \stackrel{a}{\not\Longrightarrow}$

implies B_2 **after** σ **refuses** A

Changing rôles of B_1 and B_2 proves the converse, and together they prove B_1 **after** σ **refuses** A iff B_2 **after** σ **refuses** A .

$\approx_{te} \subseteq \approx_{tr}$: Directly from theorem 3.6 and 3.5.5.

$\approx \neq \approx_{te}$: Consider $B_1 = a; (\mathbf{i}; b; \mathbf{stop} \,\Box\, \mathbf{i}; c; \mathbf{stop})$ and $B_2 = a; b; \mathbf{stop} \,\Box\, a; c; \mathbf{stop}$. $B_1 \not\approx B_2$, but $B_1 \approx_{te} B_2$.

$\approx_{te} \neq \approx_{tr}$: Consider $B_1 = a; (b; \mathbf{stop} \,\Box\, c; \mathbf{stop})$ and $B_2 = a; b; \mathbf{stop} \,\Box\, a; c; \mathbf{stop}$. $B_1 \not\approx_{te} B_2$, but $B_1 \approx_{tr} B_2$.

□

B.3.2 Section 3.3 (Implementation Relations)

Theorem 3.9

$$
\begin{array}{lll}
I \leq_{te} S & \text{iff} & \forall t \in \mathcal{LTS} : Obs(t, I) \subseteq Obs(t, S) \\
& \text{iff} & \forall t \in \mathcal{LT}_M : Obs(t, I) \subseteq Obs(t, S) \\
& \text{iff} & \forall \sigma \in L^*, \forall A \subseteq L : \\
& & \quad I \textbf{ after } \sigma \textbf{ refuses } A \quad \text{implies} \quad S \textbf{ after } \sigma \textbf{ refuses } A \\
& \text{iff} & \forall \sigma \in L^*, \forall A \subseteq L : \\
& & \quad S \textbf{ after } \sigma \textbf{ must } A \quad \text{implies} \quad I \textbf{ after } \sigma \textbf{ must } A
\end{array}
$$

□

Proof (theorem 3.9)
The first part is proved in equation (1) in the proof of proposition 3.3.1; the second part is proved in equation (1) in the proof of proposition 3.3.2.

The third and fourth part are analogous to the proof of theorem 3.6; **after** σ **must** A and **after** σ **refuses** A can be exchanged using proposition 3.5.1:

only if: Let $\sigma \in L^*$, $A \subseteq L$, then

$$
\begin{array}{ll}
 & I \textbf{ after } \sigma \textbf{ refuses } A \\
\text{iff} & \sigma \in Obs(t_{[\sigma,A]}, I) \\
\text{implies} & \sigma \in Obs(t_{[\sigma,A]}, S) \\
\text{iff} & S \textbf{ after } \sigma \textbf{ refuses } A
\end{array}
$$

if: Let $t \in \mathcal{LT}_M$, then

$$
\begin{array}{ll}
 & \sigma \in Obs(t, I) \\
\text{iff} & \exists t' \ (\ t \stackrel{\sigma}{\Longrightarrow} t' \ \text{ and } \ I \textbf{ after } \sigma \textbf{ refuses } out(t') \ ) \\
\text{implies} & \exists t' \ (\ t \stackrel{\sigma}{\Longrightarrow} t' \ \text{ and } \ S \textbf{ after } \sigma \textbf{ refuses } out(t') \ ) \\
\text{iff} & \sigma \in Obs(t, S)
\end{array}
$$

□

Proposition 3.10

1. $\approx_{te} \ = \ \leq_{te} \cap \leq_{te}^{-1}$
2. $\leq_{te}$ is a preorder.
3. $\leq_{te} \subseteq \leq_{tr}$

□

Proof (proposition 3.10)

1. Directly from theorems 3.6 and 3.9.
2. Directly from its definition.
3. Directly from theorem 3.9 and proposition 3.11.

□

Proposition 3.11
$I \leq_{tr} S$ iff $\forall \sigma \notin traces(S), \forall A \subseteq L : S \textbf{ after } \sigma \textbf{ must } A$ implies $I \textbf{ after } \sigma \textbf{ must } A$

□

Proof (proposition 3.11)

$\forall \sigma \notin \mathit{traces}(S), \forall A \subseteq L :$
S **after** σ **must** A implies I **after** σ **must** A

iff $(*$ propositions 3.5.5 and 3.5.3 $*)$
$\forall \sigma \notin \mathit{traces}(S), \forall A \subseteq L :$ I **after** σ **must** A

iff $(*$ propositions 3.5.3 $*)$
$\forall \sigma \notin \mathit{traces}(S) :$ I **after** σ **must** $\emptyset$

iff $(*$ propositions 3.5.5 $*)$
$\forall \sigma \notin \mathit{traces}(S) : \sigma \notin \mathit{traces}(I)$

iff $\mathit{traces}(I) \subseteq \mathit{traces}(S)$

iff $I \leq_{tr} S$

□

Proposition 3.13

1. $\leq_{te} \; = \; \leq_{tr} \cap$ **conf**
2. **conf** is reflexive, but not transitive.

□

Proof (proposition 3.13)

1. Directly from definition 3.12, theorem 3.9, and proposition 3.11.
2. That **conf** is reflexive, follows directly from its definition;
intransitivity of **conf** follows from e.g. $B_1 = a; \mathbf{stop} \,\Box\, c; \mathbf{stop}$, $B_2 = c; \mathbf{stop}$, and $B_3 = a; b; \mathbf{stop} \,\Box\, \mathbf{i}; c; \mathbf{stop}$: B_1 **conf** B_2, B_2 **conf** B_3, but B_1 $\not$**conf** B_3.

□

Proposition 3.14

I **conf** S iff $\forall t \in \mathcal{LTS} :$ $(\; \mathit{Obs}(t, I) \cap \mathit{traces}(S) \;) \subseteq \mathit{Obs}(t, S)$

iff $\forall t \in \mathcal{LTS} :$ $(\; \mathit{Obs}(t, I) \cap \mathit{traces}(S) \;) \subseteq \mathit{Obs}(t, S)$ and
$(\; \mathit{Obs}'(t, I) \cap \mathit{traces}(S) \;) \subseteq \mathit{Obs}'(t, S)$

□

Proof (proposition 3.14)

The first part is analogous to proposition 3.9:

only if: Let $t \in \mathcal{LTS}$, then

$\sigma \in \mathit{Obs}(t, I) \cap \mathit{traces}(S)$

iff $\exists t' \; (\; t \stackrel{\sigma}{\Longrightarrow} t' \;$ and $\; I$ **after** σ **refuses** $\mathit{out}(t') \;)$ and $\sigma \in \mathit{traces}(S)$

implies $\exists t' \; (\; t \stackrel{\sigma}{\Longrightarrow} t' \;$ and $\; S$ **after** σ **refuses** $\mathit{out}(t') \;)$

iff $\sigma \in \mathit{Obs}(t, S)$

if: Let $\sigma \in \mathit{traces}(S)$, $A \subseteq L$, then

I **after** σ **refuses** A and $\sigma \in \mathit{traces}(S)$

iff $\sigma \in \mathit{Obs}(t_{[\sigma,A]}, I)$ and $\sigma \in \mathit{traces}(S)$

implies $\sigma \in \mathit{Obs}(t_{[\sigma,A]}, S)$

iff S **after** σ **refuses** A

The second part follows from the fact that $(\ Obs'(t,I) \cap traces(S) \ ) \subseteq Obs'(t,S)$ always holds:

$\sigma \in \ (\ Obs'(t,I) \cap traces(S) \ )$
implies $(*$ definitions 1.7 and 3.1 $*)$
$t \stackrel{\sigma}{\Longrightarrow}$ and $I \stackrel{\sigma}{\Longrightarrow}$ and $S \stackrel{\sigma}{\Longrightarrow}$
implies $\sigma \in Obs'(t,S)$

□

Proposition 3.15
Let $I, S \in \mathcal{LTS}$, and let application of a must test $t_{[\sigma,A]} \in \mathcal{LT}_M$ to I, S be defined by:

$$apply_{\mathcal{C}}(t_{[\sigma,A]}, I) \ =_{def} \ \begin{cases} \textbf{pass} & \text{if } \sigma \notin Obs(t_{[\sigma,A]}, I) \\ & \quad \text{and } \exists a \in A : \sigma{\cdot}a \in Obs(t_{[\sigma,A]}, I) \\ \textbf{inconclusive} & \text{if } \sigma \notin Obs(t_{[\sigma,A]}, I) \\ & \quad \text{and } \forall a \in A : \sigma{\cdot}a \notin Obs(t_{[\sigma,A]}, I) \\ \textbf{fail} & \text{if } \sigma \in Obs(t_{[\sigma,A]}, I) \end{cases}$$

then

$$I \ \textbf{conf} \ S \ \text{ iff } \ \forall t \in \mathcal{LT}_M : \quad apply_{\mathcal{C}}(t,S) = \textbf{pass} \quad \text{implies} \quad apply_{\mathcal{C}}(t,I) \neq \textbf{fail}$$

□

Proof (proposition 3.15)

if: Let $\sigma \in traces(S)$, $A \subseteq L$, such that S **after** σ **must** A ,
then: $(*$ definition 3.4 $*)$
$\forall S' \in S \ \textbf{after} \ \sigma : \exists a \in A : S' \stackrel{a}{\Longrightarrow}$ and $\exists S'' : S \stackrel{\sigma}{\Longrightarrow} S''$
implies $\exists a \in A, \exists S'', S''' : S \stackrel{\sigma}{\Longrightarrow} S'' \stackrel{a}{\Longrightarrow} S'''$
implies $(*$ definitions 1.7 and 3.2 $*)$
$\exists a \in A, \exists S'', S''' : t_{[\sigma,A]} \,\|\, S \stackrel{\sigma}{\Longrightarrow} t_{[\epsilon,A]} \,\|\, S'' \stackrel{a}{\Longrightarrow} \textbf{stop} \,\|\, S'''$
and $\forall b \in L : \textbf{stop} \,\|\, S''' \stackrel{b}{\not\Longrightarrow}$
implies $(*$ definition 3.1.1 $*)$
$\exists a \in A : \ t_{[\sigma,A]} \,\|\, S \ \textbf{after} \ \sigma{\cdot}a \ \textbf{deadlocks}$
implies $(*$ definition 3.1.2 $*)$
$\exists a \in A : \ \sigma{\cdot}a \in Obs(t_{[\sigma,A]}, S)$
implies $(*$ definition $apply_{\mathcal{C}}$ $*)$
$(*$ proposition 3.5.6 applied to S **after** σ **must** A $*)$
$apply_{\mathcal{C}}(t_{[\sigma,A]}, S) = \textbf{pass}$
implies $(*$ premiss $*)$
$apply_{\mathcal{C}}(t_{[\sigma,A]}, I) \neq \textbf{fail}$
implies $(*$ definition $apply_{\mathcal{C}}$ $*)$
$\sigma \notin Obs(t_{[\sigma,A]}, I)$
implies $(*$ proposition 3.5.6 $*)$
I **after** σ **must** A

only if: Let $\sigma \in L^*$, $A \subseteq L$, such that $t_{[\sigma,A]} \in \mathcal{LT}_M$, then

$apply_{\mathcal{C}}(t_{[\sigma,A]}, S) = \mathbf{pass}$
iff ($*$ definition $apply_{\mathcal{C}}$ $*$)
$\sigma \notin Obs(t_{[\sigma,A]}, S)$ and $\exists a \in A : \sigma{\cdot}a \in Obs(t_{[\sigma,A]}, S)$
implies ($*$ proposition 3.5.6 and definition 3.1.2 $*$)
S **after** σ **must** A and
$\exists a \in A :\ t_{[\sigma,A]} \,\|\, S$ **after** $\sigma{\cdot}a$ **deadlocks**
implies ($*$ definition 3.1.1 $*$)
S **after** σ **must** A and $\exists a \in A : t_{[\sigma,A]} \,\|\, S \stackrel{\sigma\cdot a}{\Longrightarrow}$
implies ($*$ definition 1.7 $*$)
S **after** σ **must** A and $\sigma \in traces(S)$
implies ($*$ premiss $*$)
I **after** σ **must** A
implies ($*$ proposition 3.5.6 $*$)
$\sigma \notin Obs(t_{[\sigma,A]}, I)$
implies $apply_{\mathcal{C}}(t_{[\sigma,A]}, I) \neq \mathbf{fail}$

□

B.3.3 Section 3.4 (The Conformance Relation CONF)

Proposition 3.17
The implementation relation **conf** is compatible with the requirement language $\mathcal{L}_{must}$, specification relation $\mathbf{spec}_{\mathcal{C}}$, and satisfaction relation $\mathbf{sat}_{\mathcal{C}}$:

$$I \ \mathbf{conf}\ S \quad \text{iff} \quad I \ \mathbf{sat}_{\mathcal{C}}\ specs_{\mathcal{C}}(S)$$

□

Proof (proposition 3.17)
I **conf** S
iff $\forall \sigma \in traces(S), \forall A \subseteq L :$
S **after** σ **must** A implies I **after** σ **must** A
iff $\forall \sigma \in L^*, \forall A \subseteq L :$
($S \stackrel{\sigma}{\Longrightarrow}$ and S **after** σ **must** A) implies I **after** σ **must** A
iff $\forall \sigma \in L^*, \forall A \subseteq L :$
$S\ \mathbf{spec}_{\mathcal{C}}$ **after** σ **must** A implies $I\ \mathbf{sat}_{\mathcal{C}}$ **after** σ **must** A
iff $\forall r \in \mathcal{L}_{must} : S\ \mathbf{spec}_{\mathcal{C}}\ r$ implies $I\ \mathbf{sat}_{\mathcal{C}}\ r$
iff $\forall r \in specs_{\mathcal{C}}(S) : I\ \mathbf{sat}_{\mathcal{C}}\ r$
iff $I\ \mathbf{sat}_{\mathcal{C}}\ specs_{\mathcal{C}}(S)$

□

Proposition 3.22
Let $testreqs_{\mathcal{N}} : \mathcal{LTS} \rightarrow \mathcal{P}(\mathcal{L}_{must})$ be defined by

$$testreqs_{\mathcal{N}}(t) \ =_{def}\ \{\, \mathbf{after}\ \sigma\ \mathbf{must}\ A \mid t\ \mathbf{after}\ \sigma\ \mathbf{must}\ L \text{ and } \exists t' \,(\, t \stackrel{\sigma}{\Longrightarrow} t' \text{ and } out(t') \subseteq A \,)\, \}$$

then

$$I\ \mathbf{passes}_{\mathcal{N}}\ t \quad \text{iff} \quad I\ \mathbf{sat}_{\mathcal{C}}\ testreqs_{\mathcal{N}}(t)$$

□

Proof (proposition 3.22)

only if: Let $\sigma \in L^*, A \subseteq L$, such that **after** σ **must** $A \in \mathit{testreqs}_{\mathcal{N}}(t)$, then

$$
\begin{array}{ll}
 & \textbf{after } \sigma \textbf{ must } A \in \mathit{testreqs}_{\mathcal{N}}(t) \\
\text{implies} & (*\ \text{definition } \mathit{testreqs}_{\mathcal{N}}(t)\ *) \\
 & t \textbf{ after } \sigma \textbf{ must } L \ \text{ and } \ \exists t' \ (\ t \overset{\sigma}{\Longrightarrow} t' \ \text{ and } \ \mathit{out}(t') \subseteq A\ ) \\
\text{implies} & (*\ \text{propositions 3.5.1 and 3.5.4}\ *) \\
 & \text{not } (\ t \textbf{ after } \sigma \textbf{ deadlocks}\ ) \text{ and} \\
 & \exists t' \ (\ t \overset{\sigma}{\Longrightarrow} t' \ \text{ and } \ \mathit{out}(t') \subseteq A\ ) \\
\text{implies} & (*\ \text{premiss}\ *) \\
 & \sigma \notin \mathit{Obs}(t, I) \ \text{ and } \ \exists t' \ (\ t \overset{\sigma}{\Longrightarrow} t' \ \text{ and } \ \mathit{out}(t') \subseteq A\ ) \\
\text{implies} & (*\ \text{lemma B.1.1}\ *) \\
 & \forall t' \ (\ t \overset{\sigma}{\Longrightarrow} t' \ \text{ implies } \ I \textbf{ after } \sigma \textbf{ must } \mathit{out}(t')\ ) \ \text{ and} \\
 & \exists t' \ (\ t \overset{\sigma}{\Longrightarrow} t' \ \text{ and } \ \mathit{out}(t') \subseteq A\ ) \\
\text{implies} & \exists t' \ (\ t \overset{\sigma}{\Longrightarrow} t' \ \text{ and } \ \mathit{out}(t') \subseteq A \ \text{ and } \ I \textbf{ after } \sigma \textbf{ must } \mathit{out}(t')\ ) \\
\text{implies} & (*\ \text{proposition 3.5.3}\ *) \\
 & I \textbf{ after } \sigma \textbf{ must } A \\
\text{iff} & I \textbf{ sat}_{\mathcal{C}} \ \textbf{after } \sigma \textbf{ must } A
\end{array}
$$

if: By contraposition:

$$
\begin{array}{ll}
 & I \ \cancel{\textbf{passes}}_{\mathcal{N}} \ t \\
\text{implies} & (*\ \text{definition 3.20}\ *) \\
 & \exists \sigma \in L^* : \sigma \in \mathit{Obs}(t, I) \ \text{ and } \ t \textbf{ after } \sigma \textbf{ must } L \\
\text{implies} & (*\ \text{lemma B.1.1}\ *) \\
 & \exists \sigma \in L^* : \exists t' \ (\ t \overset{\sigma}{\Longrightarrow} t' \ \text{ and } \ I \textbf{ after } \sigma \textbf{ refuses } \mathit{out}(t')\ ) \\
 & \quad \text{and } \ t \textbf{ after } \sigma \textbf{ must } L \\
\text{implies} & (*\ \text{definition } \mathit{testreqs}_{\mathcal{N}}(t)\ *) \\
 & \exists \sigma \in L^*, \exists t' : \ \textbf{after } \sigma \textbf{ must } \mathit{out}(t') \in \mathit{testreqs}_{\mathcal{N}}(t) \\
 & \quad \text{and not } (\ I \textbf{ after } \sigma \textbf{ must } \mathit{out}(t')\ ) \\
\text{implies} & \exists r = \textbf{after } \sigma \textbf{ must } \mathit{out}(t') \in \mathit{testreqs}_{\mathcal{N}}(t) : I \ \cancel{\textbf{sat}}_{\mathcal{C}} \ r
\end{array}
$$

□

Proposition 3.23
Let $S \in \mathcal{LTS}$, then the test suite

$$\Pi^{tr}_{\textbf{conf}}(S) \ =_{def} \ \{\ t \in \mathcal{LTS} \mid \mathit{traces}(t) \subseteq \mathit{traces}(S), \ S \textbf{ passes}_{\mathcal{N}} \ t\ \}$$

is complete for **conf**-conforming implementations of S, i.e.

$$\forall I \in \mathcal{LTS} : \ I \textbf{ conf } S \quad \text{iff} \quad I \textbf{ passes}_{\mathcal{N}} \ \Pi^{tr}_{\textbf{conf}}(S)$$

□

Proof (proposition 3.23)

soundness: Let $I \in \mathcal{LTS}$, then we prove, using proposition 3.22:

$$I \textbf{ conf } S \quad \text{implies} \quad \forall t \in \Pi^{tr}_{\textbf{conf}}(S), \forall r \in \mathit{testreqs}_{\mathcal{N}}(t) : I \textbf{ sat}_{\mathcal{C}} r$$

Let $t \in \Pi^{tr}_{\textbf{conf}}(S)$, and $r = \textbf{after } \sigma \textbf{ must } A \in \mathit{testreqs}_{\mathcal{N}}(t)$, then

$\mathit{traces}(t) \subseteq \mathit{traces}(S)$ and $S \textbf{ passes}_{\mathcal{N}} t$
and $\exists t' \; (\; t \stackrel{\sigma}{\Longrightarrow} t' \text{ and } \mathit{out}(t') \subseteq A \;)$

implies $(* \text{ proposition 3.22 applied to } S \; *)$
$\sigma \in \mathit{traces}(S)$ and $S \textbf{ after } \sigma \textbf{ must } A$

implies $(* \text{ premiss } *)$
$I \textbf{ after } \sigma \textbf{ must } A$

iff $I \textbf{ sat}_{\mathcal{C}} r$

exhaustiveness: We prove: $\forall r \in \mathcal{L}_{must} : S \textbf{ spec}_{\mathcal{C}} r$ implies $I \textbf{ sat}_{\mathcal{C}} r$.

We use a modified version of a must test: $u_{[\sigma,A]}$ is a must test with for all states that are reached with $\sigma' \prec \sigma$ an extra branch $\tau; \textbf{stop}$.

Let $r = \textbf{after } \sigma \textbf{ must } A \in \mathcal{L}_{must}$, then

$S \textbf{ spec}_{\mathcal{C}} r$

implies $S \stackrel{\sigma}{\Longrightarrow}$ and $S \textbf{ after } \sigma \textbf{ must } A$

implies $\exists S' : S \stackrel{\sigma}{\Longrightarrow} S'$ and $\forall S'' \; (\; S \stackrel{\sigma}{\Longrightarrow} S'' \text{ implies } \exists a \in A : S'' \stackrel{a}{\Longrightarrow})$

implies $(* \text{ proposition 3.5.5 } *)$
$A \neq \emptyset$ and $A' = \{ \; a \in L \mid S \stackrel{\sigma \cdot a}{\Longrightarrow} \} \neq \emptyset$ and $A \cap A' \neq \emptyset$

implies $\mathit{traces}(u_{[\sigma,A\cap A']}) \subseteq \mathit{traces}(S)$ and
$\forall S'' \; (\; S \stackrel{\sigma}{\Longrightarrow} S'' \text{ implies } \exists a \in A \cap A' : S'' \stackrel{a}{\Longrightarrow})$

implies $\mathit{traces}(u_{[\sigma,A\cap A']}) \subseteq \mathit{traces}(S)$ and $S \textbf{ after } \sigma \textbf{ must } A \cap A'$

implies $(* \text{ proposition 3.22 } *)$
$\mathit{traces}(u_{[\sigma,A\cap A']}) \subseteq \mathit{traces}(S)$ and $S \textbf{ passes}_{\mathcal{N}} u_{[\sigma,A\cap A']}$

implies $(* \text{ definition } \Pi^{tr}_{\textbf{conf}}(S) \; *)$
$u_{[\sigma,A\cap A']} \in \Pi^{tr}_{\textbf{conf}}(S)$

implies $(* \text{ premiss } *)$
$I \textbf{ passes}_{\mathcal{N}} u_{[\sigma,A\cap A']}$

implies $(* \text{ proposition 3.22.2 } *)$
$I \textbf{ sat}_{\mathcal{C}} \textbf{ after } \sigma \textbf{ must } A \cap A'$

implies $(* \text{ proposition 3.5.3 } *)$
$I \textbf{ sat}_{\mathcal{C}} \textbf{ after } \sigma \textbf{ must } A$

iff $I \textbf{ sat}_{\mathcal{C}} r$

□

Proposition 3.26
Let $\mathit{testreqs}_{\mathcal{D}} : \mathcal{DLTS} \rightarrow \mathcal{P}(\mathcal{L}_{must})$ be defined by

$$\mathit{testreqs}_{\mathcal{D}}(t) \; =_{def} \; \{ \textbf{after } \sigma \textbf{ must } A \mid v(\, t \textbf{ after } \sigma \,) = \textbf{fail} \text{ and } \mathit{out}(\, t \textbf{ after } \sigma \,) \subseteq A \}$$

then $$I \textbf{ passes}_{\mathcal{D}} t \quad \text{iff} \quad I \textbf{ sat}_{\mathcal{C}} \mathit{testreqs}_{\mathcal{D}}(t)$$

□

Proof (proposition 3.26)
Analogous to the proof of proposition 3.22:

only if: Let $\sigma \in L^*, A \subseteq L$, such that $\quad \textbf{after}\ \sigma\ \textbf{must}\ A\ \in \mathit{testreqs}_{\mathcal{D}}(t)$, then
$\textbf{after}\ \sigma\ \textbf{must}\ A\ \in \mathit{testreqs}_{\mathcal{D}}(t)$
implies $\quad (*\ \text{definition}\ \mathit{testreqs}_{\mathcal{D}}(t)\ *)$
$v(\, t\ \textbf{after}\ \sigma\,) = \textbf{fail}\ \text{ and }\ \mathit{out}(\, t\ \textbf{after}\ \sigma\,) \subseteq A$
implies $\quad (*\ \text{premiss}\ *)$
$\sigma \notin \mathit{Obs}(t, I)\ \text{ and }\ t \stackrel{\sigma}{\Longrightarrow} t\ \textbf{after}\ \sigma\ \text{ and }\ \mathit{out}(\, t\ \textbf{after}\ \sigma\,) \subseteq A$
implies $\quad (*\ \text{lemma B.1.1}\ *)$
$\forall t'\ (\ t \stackrel{\sigma}{\Longrightarrow} t'\ \text{ implies }\ I\ \textbf{after}\ \sigma\ \textbf{must}\ \mathit{out}(t')\ )$
$\text{and }\ t \stackrel{\sigma}{\Longrightarrow} t\ \textbf{after}\ \sigma\ \text{ and }\ \mathit{out}(\, t\ \textbf{after}\ \sigma\,) \subseteq A$
implies $\quad I\ \textbf{after}\ \sigma\ \textbf{must}\ \mathit{out}(\, t\ \textbf{after}\ \sigma\,)\ \text{ and }\ \mathit{out}(\, t\ \textbf{after}\ \sigma\,) \subseteq A$
implies $\quad I\ \textbf{after}\ \sigma\ \textbf{must}\ A$
iff $\quad I\ \textbf{sat}_{\mathcal{C}}\ \textbf{after}\ \sigma\ \textbf{must}\ A$

if: By contraposition:
$I\ \not{\textbf{passes}}_{\mathcal{D}}\ t$
implies $\quad (*\ \text{definition 3.25.2}\ *)$
$\exists \sigma \in L^* : \sigma \in \mathit{Obs}(t, I)\ \text{ and }\ v(\, t\ \textbf{after}\ \sigma\,) = \textbf{fail}$
implies $\quad (*\ \text{lemma B.1.1}\ *)$
$\exists \sigma \in L^* : \exists t'\ (\ t \stackrel{\sigma}{\Longrightarrow} t'\ \text{ and }\ I\ \textbf{after}\ \sigma\ \textbf{refuses}\ \mathit{out}(t')\ )$
$\text{and }\ v(\, t\ \textbf{after}\ \sigma\,) = \textbf{fail}$
implies $\quad \exists \sigma \in L^* :\ I\ \textbf{after}\ \sigma\ \textbf{refuses}\ \mathit{out}(\, t\ \textbf{after}\ \sigma\,)$
$\text{and }\ v(\, t\ \textbf{after}\ \sigma\,) = \textbf{fail}$
implies $\quad \exists r = \ \textbf{after}\ \sigma\ \textbf{must}\ \mathit{out}(\, t\ \textbf{after}\ \sigma\,)\ \in \mathit{testreqs}_{\mathcal{D}}(t) :\ I\ \not{\textbf{sat}}_{\mathcal{C}}\ r$ □

Proposition 3.27
Let $S \in \mathcal{LTS}$, then the test suite

- $\Pi^{det}_{\textbf{conf}}(S)\ =_{def}\ \{\ t \in \mathcal{DLTS} \mid \mathit{traces}(t) \subseteq \mathit{traces}(S),\ S\ \textbf{passes}_{\mathcal{D}}\ t\ \}$

is complete for **conf**-conforming implementations of S. □

Proof (proposition 3.27)
Analogous to the proof of proposition 3.23.2:

soundness: Let $I \in \mathcal{LTS}$, then we prove, using proposition 3.26:
$I\ \textbf{conf}\ S\ \quad \text{implies}\ \quad \forall t \in \Pi^{det}_{\textbf{conf}}(S), \forall r \in \mathit{testreqs}_{\mathcal{D}}(t) : I\ \textbf{sat}_{\mathcal{C}}\ r$
Let $t \in \Pi^{det}_{\textbf{conf}}(S)$, and $r = \ \textbf{after}\ \sigma\ \textbf{must}\ A\ \in \mathit{testreqs}_{\mathcal{D}}(t)$, then
$\mathit{traces}(t) \subseteq \mathit{traces}(S)\ \text{ and }\ S\ \textbf{passes}_{\mathcal{D}}\ t$
$\text{and }\ v(\, t\ \textbf{after}\ \sigma\,) = \textbf{fail}\ \text{ and }\ \mathit{out}(\, t\ \textbf{after}\ \sigma\,) \subseteq A$
implies $\quad (*\ \text{proposition 3.26 applied to}\ S\ *)$
$\sigma \in \mathit{traces}(S)\ \text{ and }\ S\ \textbf{after}\ \sigma\ \textbf{must}\ A$
implies $\quad (*\ \text{premiss}\ *)$
$I\ \textbf{after}\ \sigma\ \textbf{must}\ A$
iff $\quad I\ \textbf{sat}_{\mathcal{C}}\ r$

exhaustiveness: We prove: $\forall r \in \mathcal{L}_{must} : S \ \mathbf{spec}_{\mathcal{C}} \ r$ implies $I \ \mathbf{sat}_{\mathcal{C}} \ r$.

Let $r = \ \mathbf{after} \ \sigma \ \mathbf{must} \ A \ \in \mathcal{L}_{must}$, then

$S \ \mathbf{spec}_{\mathcal{C}} \ r$

implies $S \stackrel{\sigma}{\Longrightarrow}$ and $S \ \mathbf{after} \ \sigma \ \mathbf{must} \ A$

implies $A \neq \emptyset$ and $A' = \{ \ a \in L \mid S \stackrel{\sigma \cdot a}{\Longrightarrow} \} \neq \emptyset$ and $A \cap A' \neq \emptyset$

Let $\sigma = b_1 \cdot b_2 \cdot \ldots \cdot b_m$, and let $u \in \mathcal{DLTS}$ be:

$$u \ =_{def} \ b_1; b_2; \ldots; b_m; \Sigma\{a; \mathbf{stop} \mid a \in A \cap A'\}$$

with $v(\ u \ \mathbf{after} \ \sigma' \ ) = \mathbf{fail}$ iff $\sigma' = \sigma$,

then $testreqs_{\mathcal{D}}(u) = \{ \ \mathbf{after} \ \sigma \ \mathbf{must} \ A'' \ \mid A \cap A' \subseteq A'' \ \}$, and it follows that:

$traces(u) \subseteq traces(S)$ and $S \ \mathbf{after} \ \sigma \ \mathbf{must} \ A \cap A'$

implies $traces(u) \subseteq traces(S)$ and $S \ \mathbf{sat}_{\mathcal{C}} \ testreqs_{\mathcal{D}}(u)$

implies $(*$ proposition 3.26 $*)$

$traces(u) \subseteq traces(S)$ and $S \ \mathbf{passes}_{\mathcal{D}} \ u$

implies $u \in \Pi^{det}_{\mathbf{conf}}(S)$

implies $(*$ premiss: $I \ \mathbf{passes}_{\mathcal{D}} \ \Pi^{det}_{\mathbf{conf}}(S) \ *)$

$I \ \mathbf{passes}_{\mathcal{D}} \ u$

implies $(*$ proposition 3.26 $*)$

$\forall A'' \supseteq A \cap A' : I \ \mathbf{sat}_{\mathcal{C}} \ \mathbf{after} \ \sigma \ \mathbf{must} \ A''$

implies $I \ \mathbf{sat}_{\mathcal{C}} \ \mathbf{after} \ \sigma \ \mathbf{must} \ A$

iff $I \ \mathbf{sat}_{\mathcal{C}} \ r$

□

Proposition 3.31

Derivation for **conf**-theories T over the class of models $\mathcal{LTS}$ with the satisfaction relation $\mathbf{sat}_{\mathcal{C}}$ is sound and complete, i.e.

$$T \vdash r \quad \text{iff} \quad \forall I \in \mathcal{LTS} : I \ \mathbf{sat}_{\mathcal{C}} \ T \ \text{ implies } \ I \ \mathbf{sat}_{\mathcal{C}} \ r$$

□

Proof (proposition 3.31)

Soundness: Let $I \in \mathcal{LTS}$, with $I \ \mathbf{sat}_{\mathcal{C}} \ T$, then

$T \vdash \ \mathbf{after} \ \sigma \ \mathbf{must} \ A$

implies $(*$ definition 3.30 $*)$

$\exists A' \subseteq A : \ \mathbf{after} \ \sigma \ \mathbf{must} \ A' \ \in T$

implies $\exists A' \subseteq A : I \ \mathbf{sat}_{\mathcal{C}} \ \mathbf{after} \ \sigma \ \mathbf{must} \ A'$

implies $\exists A' \subseteq A : \ I \ \mathbf{after} \ \sigma \ \mathbf{must} \ A'$

implies $I \ \mathbf{after} \ \sigma \ \mathbf{must} \ A$

implies $I \ \mathbf{sat}_{\mathcal{C}} \ \mathbf{after} \ \sigma \ \mathbf{must} \ A$

Completeness: By contraposition. Let $\sigma \in L^*$, $A \subseteq L$, then we have to prove:

$T \not\vdash \ \mathbf{after} \ \sigma \ \mathbf{must} \ A$ implies $\exists I : I \ \mathbf{sat}_{\mathcal{C}} \ T$ and $I \not\mathbf{sat}_{\mathcal{C}} \ \mathbf{after} \ \sigma \ \mathbf{must} \ A$

From the definition of $\vdash$:

$$T \not\vdash \ \mathbf{after} \ \sigma \ \mathbf{must} \ A \quad \text{iff} \quad \forall A' \subseteq A : \ \mathbf{after} \ \sigma \ \mathbf{must} \ A' \ \notin T \tag{1}$$

Let I_L be the process that can always perform any action:

$$I_L \;=_{def}\; \Sigma\{\; a; I_L \mid a \in L \;\}$$

and let I'_L be I_L with the modification that the state $I_L \,\mathbf{after}\, \sigma$, which can be written as $I_L \,\mathbf{after}\, \sigma = \Sigma\{a; I_L \mid a \in L\}$, is replaced by $\Sigma\{a; I_L \mid a \in A\} \,\Box\, \mathbf{i}; \Sigma\{a; I_L \mid a \in L \backslash A\}$, then I'_L has the required properties:

I'_L $\mathbf{sat}_{\mathcal{C}}$ T: I_L $\mathbf{sat}_{\mathcal{C}}$ **after** σ **must** A for any $\sigma \in L^*$ and $\emptyset \neq A \subseteq L$. For I'_L this is changed only for all requirements **after** σ **must** A' with $\emptyset \neq A' \subseteq A$. These are exactly those requirements which are not in T according to (1), hence I'_L $\mathbf{sat}_{\mathcal{C}}$ T.

I'_L $\not{\mathbf{sat}}_{\mathcal{C}}$ **after** σ **must** A : $I'_L \stackrel{\sigma}{\Longrightarrow} \Sigma\{a; I_L \mid a \in L \backslash A\}$ and $\forall a \in A : \Sigma\{a; I_L \mid a \in L \backslash A\} \stackrel{a}{\not\Longrightarrow}$. Hence, I'_L **after** σ **refuses** A .

□

B.4 Chapter 4 (Synchronous Testing)

B.4.1 Section 4.2 (Test Derivation for Labelled Transition Systems)

Proposition 4.3
Let $S \in \mathcal{LTS}$, $\sigma, \sigma_1, \sigma_2 \in L^*$, $A \subseteq L$, then

1. **choice** S **after** σ $\quad =_{def} \quad \Sigma\{\ \mathbf{i}; S' \mid S \stackrel{\sigma}{\Longrightarrow} S'\ \}$
   $\quad \approx_{te} \quad \Sigma\{\ \mathbf{i}; S' \mid S \stackrel{\sigma}{\Longrightarrow} S'\ \}$
2. **choice** S **after** ϵ $\approx_{te}$ S
3. For $\sigma_1 \in traces(S)$ or $\sigma_2 \neq \epsilon$:

$$S \textbf{ after } \sigma_1 \cdot \sigma_2 \textbf{ must } A \quad \text{iff} \quad (\textbf{ choice } S \textbf{ after } \sigma_1) \textbf{ after } \sigma_2 \textbf{ must } A$$

4. **choice** S **after** $\sigma_1 \cdot \sigma_2$ $\approx_{te}$ **choice** (**choice** S **after** σ_1) **after** σ_2

□

Proof (proposition 4.3)

1. Let $U = \{S' \mid S \stackrel{\sigma}{\Longrightarrow} S'\}$, then, since $S \stackrel{\sigma}{\Longrightarrow} S'$ iff $\exists S'' : S \stackrel{\sigma}{\Longrightarrow} S'' \stackrel{\epsilon}{\Longrightarrow} S'$, the proposition is stated as:

$$\Sigma\{\ \mathbf{i}; S' \mid \exists S'' \in U : S'' \stackrel{\epsilon}{\Longrightarrow} S'\ \} \approx_{te} \Sigma\{\ \mathbf{i}; S' \mid S' \in U\ \}$$

Let $B_1 = \Sigma\{\mathbf{i}; S' \mid \exists S'' \in U : S'' \stackrel{\epsilon}{\Longrightarrow} S'\}$ and $B_2 = \Sigma\{\mathbf{i}; S' \mid S' \in U\}$, then we have to prove: $\forall \rho \in L^*, A \subseteq L^* :$ B_1 **after** ρ **refuses** A iff B_2 **after** ρ **refuses** A , which is equivalent to

$\forall \rho \in L^*, A \subseteq L^* :$
$\exists B' : B_1 \stackrel{\rho}{\Longrightarrow} B'$ and $\forall a \in A : B' \stackrel{a}{\Longrightarrow\!\!\!\!\!\!/}$ iff $\exists B' : B_2 \stackrel{\rho}{\Longrightarrow} B'$ and $\forall a \in A : B' \stackrel{a}{\Longrightarrow\!\!\!\!\!\!/}$

Let $\rho \in L^*$, $A \subseteq L$, B' such that $B_1 \stackrel{\rho}{\Longrightarrow} B'$, and distinguish between $B' = B_1$ (and $\rho = \epsilon$), and $B' \neq B_1$:

$B' = B_1$:
From the definitions of B_1 and B_2 it follows directly that
$out(B_1) = out(B_2) = out(\Sigma U)$,
hence $\forall a \in A : B_1 \stackrel{a}{\Longrightarrow\!\!\!\!\!\!/}$ iff $\forall a \in A : B_2 \stackrel{a}{\Longrightarrow\!\!\!\!\!\!/}$, and

$$\begin{aligned} & \exists B' = B_1 : B_1 \stackrel{\epsilon}{\Longrightarrow} B' \text{ and } \forall a \in A : B' \stackrel{a}{\Longrightarrow\!\!\!\!\!\!/} \\ \text{iff} \quad & \exists B' = B_2 : B_2 \stackrel{\epsilon}{\Longrightarrow} B' \text{ and } \forall a \in A : B' \stackrel{a}{\Longrightarrow\!\!\!\!\!\!/} \end{aligned}$$

$B' \neq B_1$:

$$\begin{array}{ll} & \exists B' \neq B_1 : B_1 \stackrel{\rho}{\Longrightarrow} B' \text{ and } \forall a \in A : B' \stackrel{a}{\Longrightarrow\!\!\!\!\!\!/} \\ \text{iff} & \exists B' : B_1 \stackrel{\tau}{\longrightarrow} \stackrel{\rho}{\Longrightarrow} B' \text{ and } \forall a \in A : B' \stackrel{a}{\Longrightarrow\!\!\!\!\!\!/} \\ \text{iff} & \exists B' : \Sigma\{\mathbf{i}; S' \mid \exists S'' \in U : S'' \stackrel{\epsilon}{\Longrightarrow} S'\} \stackrel{\tau}{\longrightarrow} \stackrel{\rho}{\Longrightarrow} B' \text{ and } \forall a \in A : B' \stackrel{a}{\Longrightarrow\!\!\!\!\!\!/} \\ \text{iff} & \exists B', \exists S', \exists S'' \in U : S'' \stackrel{\epsilon}{\Longrightarrow} S' \text{ and } S' \stackrel{\rho}{\Longrightarrow} B' \text{ and } \forall a \in A : B' \stackrel{a}{\Longrightarrow\!\!\!\!\!\!/} \\ \text{iff} & \exists B', \exists S'' \in U : S'' \stackrel{\rho}{\Longrightarrow} B' \text{ and } \forall a \in A : B' \stackrel{a}{\Longrightarrow\!\!\!\!\!\!/} \\ \text{iff} & \exists B' : \Sigma\{\mathbf{i}; S' \mid S' \in U\} \stackrel{\tau}{\longrightarrow} \stackrel{\rho}{\Longrightarrow} B' \text{ and } \forall a \in A : B' \stackrel{a}{\Longrightarrow\!\!\!\!\!\!/} \\ \text{iff} & \exists B' : B_2 \stackrel{\tau}{\longrightarrow} \stackrel{\rho}{\Longrightarrow} B' \text{ and } \forall a \in A : B' \stackrel{a}{\Longrightarrow\!\!\!\!\!\!/} \\ \text{iff} & \exists B' \neq B_2 : B_2 \stackrel{\rho}{\Longrightarrow} B' \text{ and } \forall a \in A : B' \stackrel{a}{\Longrightarrow\!\!\!\!\!\!/} \end{array}$$

2. $\quad$ **choice** S **after** ϵ

$$\begin{array}{ll} \approx_{te} & \Sigma\{\mathbf{i}; S' \mid S \stackrel{\epsilon}{\longrightarrow} S'\} \\ = & \mathbf{i}; S \\ \approx_{te} & S \end{array}$$

For 3. and 4. we first prove for $\sigma_1 \in \mathit{traces}(S_1)$ or $\sigma_2 \neq \epsilon$:

$$S \textbf{ after } \sigma_1 \cdot \sigma_2 \;=\; (\textbf{ choice } S \textbf{ after } \sigma_1) \textbf{ after } \sigma_2 \qquad (1)$$

$$\begin{array}{ll} & S_1 \in S \textbf{ after } \sigma_1 \cdot \sigma_2 \\ \text{iff} & S \stackrel{\sigma_1 \cdot \sigma_2}{\Longrightarrow} S_1 \\ \text{iff} & \exists S_2 : S \stackrel{\sigma_1}{\Longrightarrow} S_2 \stackrel{\sigma_2}{\Longrightarrow} S_1 \\ \text{iff} & \exists S_2 : \Sigma\{\mathbf{i}; S' \mid S \stackrel{\sigma_1}{\Longrightarrow} S'\} \stackrel{\tau}{\longrightarrow} S_2 \stackrel{\sigma_2}{\Longrightarrow} S_1 \\ \text{iff} & (* \text{ for } \textit{if} \text{ use } \sigma_1 \in \mathit{traces}(S) \text{ or } \sigma_2 \neq \epsilon \; *) \\ & \Sigma\{\mathbf{i}; S' \mid S \stackrel{\sigma_1}{\Longrightarrow} S'\} \stackrel{\sigma_2}{\Longrightarrow} S_1 \\ \text{iff} & S_1 \in (\textbf{ choice } S \textbf{ after } \sigma_1) \textbf{ after } \sigma_2 \end{array}$$

Now the proofs of 3. and 4.:

3. Directly from (1).
4. If $\sigma_2 \neq \epsilon$ directly from (1). For $\sigma_2 = \epsilon$ directly from proposition 4.3.2.

□

Lemma B.2
Let the test suite $\Pi_R(S) \subseteq \mathcal{DLTS}$ be defined, for $S \in \mathcal{LTS}$, by

$$\begin{array}{lll} \Pi_R(S) & =_{def} & \{\, t \in \mathcal{DLTS} \mid \; t = \Sigma \{\, a; t_a \mid a \in A \,\}, \\ & & \quad A \subseteq \mathit{out}(S), \\ & & \quad v(t) = \textbf{fail} \quad \text{implies} \quad S \textbf{ after } \epsilon \textbf{ must } A\,, \\ & & \quad t_a \in \Pi_R(\textbf{ choice } S \textbf{ after } a) \,\}, \end{array}$$

then $\Pi_R(S)$ is complete.

□

Proof (lemma B.2)
We prove, using equation (4.1),

$$\forall S \in \mathcal{LTS} : \mathit{testreqs}_{\mathcal{D}}(\Pi_R(S)) = \mathit{specs}_{\mathcal{C}}(S)$$

which is equivalent to

$$\begin{array}{l}\forall S \in \mathcal{LTS}, \forall \sigma \in L^*, \forall A \subseteq L : \\ \qquad \exists t \in \Pi_R(S) : v(\, t \,\mathbf{after}\, \sigma\,) = \mathbf{fail} \;\text{ and }\; out(\, t \,\mathbf{after}\, \sigma\,) \subseteq A \\ \quad \text{iff} \quad S \stackrel{\sigma}{\Longrightarrow} \;\text{ and }\; S \,\mathbf{after}\, \sigma \,\mathbf{must}\, A\end{array}$$

by induction on the length of σ:

ϵ: To prove: $\forall S \in \mathcal{LTS}, \forall A \subseteq L : \;\; \exists t \in \Pi_R(S) : v(t) = \mathbf{fail} \;\text{ and }\; out(t) \subseteq A$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{iff} \quad S \stackrel{\epsilon}{\Longrightarrow} \;\text{ and }\; S \,\mathbf{after}\, \epsilon \,\mathbf{must}\, A$

only if: Directly from the definition of $\Pi_R(S)$, using proposition 3.5.3.

if: The test case $t = \Sigma\{a; \mathbf{stop} \mid a \in A \cap out(S)\}$ with $v(t) = \mathbf{fail}$ and $v(\mathbf{stop}) = \mathbf{pass}$ fulfils the requirements.

$a{\cdot}\sigma$: To prove:

$$\begin{array}{l}\forall S \in \mathcal{LTS}, \forall A \subseteq L : \\ \qquad \exists t \in \Pi_R(S) : v(\, t \,\mathbf{after}\, a{\cdot}\sigma\,) = \mathbf{fail} \;\text{ and }\; out(\, t \,\mathbf{after}\, a{\cdot}\sigma\,) \subseteq A \\ \quad \text{iff} \quad S \stackrel{a\cdot\sigma}{\Longrightarrow} \;\text{ and }\; S \,\mathbf{after}\, a{\cdot}\sigma \,\mathbf{must}\, A\end{array}$$

$$\begin{array}{ll} & \exists t \in \Pi_R(S) : v(\, t \,\mathbf{after}\, a{\cdot}\sigma\,) = \mathbf{fail} \;\text{ and }\; out(\, t \,\mathbf{after}\, a{\cdot}\sigma\,) \subseteq A \\ \text{iff} & (*\;\text{definition } \Pi_R(S)\;*) \\ & \exists t \in \Pi_R(S), \exists t' : t \stackrel{a}{\longrightarrow} t' \stackrel{\sigma}{\Longrightarrow} t' \,\mathbf{after}\, \sigma \\ & \;\text{and }\; v(\, t' \,\mathbf{after}\, \sigma\,) = \mathbf{fail} \;\text{ and }\; out(\, t' \,\mathbf{after}\, \sigma\,) \subseteq A \\ \text{iff} & (*\;\text{definition } \Pi_R(S),\, a \in out(t) \subseteq out(S)\;*) \\ & a \in out(S) \;\text{ and} \\ & \exists t' \in \Pi_R(\, \mathbf{choice}\, S \,\mathbf{after}\, a\,) : v(\, t' \,\mathbf{after}\, \sigma\,) = \mathbf{fail} \;\text{ and} \\ & \qquad\qquad\qquad\qquad\qquad\qquad out(\, t' \,\mathbf{after}\, \sigma\,) \subseteq A \\ \text{iff} & (*\;\text{induction hypothesis}\;*) \\ & a \in out(S) \;\text{ and }\; \mathbf{choice}\, S \,\mathbf{after}\, a \stackrel{\sigma}{\Longrightarrow} \;\text{ and} \\ & (\, \mathbf{choice}\, S \,\mathbf{after}\, a\,) \,\mathbf{after}\, \sigma \,\mathbf{must}\, A \\ \text{iff} & (*\;\text{proposition 4.3.3}\;*) \\ & S \stackrel{a\cdot\sigma}{\Longrightarrow} \;\text{ and }\; S \,\mathbf{after}\, a{\cdot}\sigma \,\mathbf{must}\, A\end{array}$$

□

Proposition 4.5
Let L be finite, and let $S \in \mathcal{LTS}$ have finite behaviour.

1. $t_A \;=_{def}\; \Sigma\{a; t_a \mid a \in A\}$ is a sound test case for S, if
   $A \subseteq out(S)$, and
   $v(t_A) = \mathbf{fail}$ implies $S \,\mathbf{after}\, \epsilon \,\mathbf{must}\, A$, and
   t_a is a sound test case for $\mathbf{choice}\, S \,\mathbf{after}\, a$.
2. The test suite
$$\begin{array}{rl}\Pi^1_{\mathbf{conf}}(S) \;=_{def}\; \{\; t_A \in \mathcal{DLTS} \mid & t_A = \Sigma\{a; t_a \mid a \in A\}, \\ & \quad (\; A \in min_{\subseteq}(\overline{M_\epsilon}(S)) \;\text{ and }\; v(t_A) = \mathbf{fail}\;) \\ & \text{or }\; (\; A = out(S) \;\text{ and }\; v(t_A) = \mathbf{pass}\;), \\ & t_a \in \Pi^1_{\mathbf{conf}}(\, \mathbf{choice}\, S \,\mathbf{after}\, a\,)\;\},\end{array}$$

is complete.

□

Proof (proposition 4.5)

1. Directly from lemma B.2.
2. Soundness directly from lemma B.2: $\Pi^1_{\mathbf{conf}}(S) \subseteq \Pi_R(S)$. Completeness follows from the fact that for each test case in $\Pi_R(S)$ there is a test case in $\Pi^1_{\mathbf{conf}}(S)$ that tests the same requirements.

□

Proposition 4.7
If $S \in \mathcal{LTS}$ is image-finite, and $\overline{M_\epsilon}(S) \neq \emptyset$, then minimal elements of $\overline{M_\epsilon}(S)$ exist.

□

Proof (proposition 4.7)
If S is image-finiteness (definition 1.11.6), then S **after** ϵ **must** A is finite, hence there exists a finite A' with S **after** ϵ **must** A', viz. $A' = \{a_1, a_2, \ldots, a_n\}$ with $a_i \in out(S_i)$ for each $S_i \in S$ **after** ϵ. If $\nexists a_i \in out(S_i)$ for some i then $\overline{M_\epsilon}(S) = \emptyset$.

This A' is an element of the set of all finite subsets of L, which well-founded, hence minimal elements exist (appendix A).

□

Proposition 4.9
$\sqsubseteq$ is a partial order.

□

Proof (proposition 4.9)
Directly from the facts that $\subseteq$ and $=$ are partial orders.

□

Proposition 4.10
Let $S \in \mathcal{LTS}$, then $\Pi \in \mathcal{P}(\mathcal{DLTS})$ is a complete test suite for S with respect to **conf**, if $\exists M \sqsubseteq \overline{M_\epsilon}(S)$:

$$
\begin{aligned}
\Pi \;=\; \{\; t_A \in \mathcal{DLTS} \mid\; & t_A = \Sigma\{a; t_a \mid a \in A\}, \\
& (\; A \in M \;\text{ and }\; v(t_A) = \mathbf{fail} \;) \\
& \text{or } (\; A = out(S) \;\text{ and }\; v(t_A) = \mathbf{pass} \;) \\
& \text{or } (\; A = \emptyset \;\text{ and }\; v(t_A) = \mathbf{pass} \;), \\
& \text{for each } a \in A : t_a \;\text{ is element of a complete test suite} \\
& \text{for } \mathbf{choice}\ S\ \mathbf{after}\ a \quad \}
\end{aligned}
$$

□

Proof (proposition 4.10)

$\Pi \subseteq \Pi_R(S)$: Let $t = \Sigma\{a; t_a \mid a \in A\} \in \Pi$ then

(1): $A \in M \sqsubseteq \overline{M_\epsilon}(S)$ and $v(t) = \mathbf{fail}$
implies $A \subseteq out(S)$ and S **after** ϵ **must** A
implies $t \in \Pi_R(S)$ iff $\forall a \in A : t_a \in \Pi_R(\,\mathbf{choice}\ S\ \mathbf{after}\ a\,)$

(2): $M = \emptyset$ and $A = out(S)$ and $v(t) = \mathbf{pass}$
implies $t \in \Pi_R(S)$ iff $\forall a \in A : t_a \in \Pi_R(\,\mathbf{choice}\ S\ \mathbf{after}\ a\,)$

(3): $A = \emptyset$ and $v(t) = \mathbf{pass}$
implies $t = \mathbf{stop} \in \Pi_R(S)$

Together they prove, by induction, with (3) forming the basis, that $t \in \Pi_R(S)$ (Strictly speaking only for t with finite behaviour).

completeness: Using proposition B.2:
$\forall S \in \mathcal{LTS} : testreqs_{\mathcal{D}}(\Pi_R(S)) = testreqs_{\mathcal{D}}($
$Pi) :$

$\subseteq$: By induction on the length of σ:

ϵ: To prove:

$$\begin{array}{l} \forall S \in \mathcal{LTS}, \forall A \subseteq L : \\ \qquad \exists t \in \Pi_R(S) : v(t) = \mathbf{fail} \text{ and } out(t) \subseteq A \\ \text{implies} \quad \exists t' \in \Pi : v(t') = \mathbf{fail} \text{ and } out(t') \subseteq A \end{array}$$

$$\begin{array}{ll} \text{Let } t = \Sigma\{b; t_b \mid b \in B\} \in \Pi_R(S), & \\ \quad \text{then} & B \subseteq out(S) \text{ and } B = out(t) \subseteq A \\ & \text{and } S \textbf{ after } \epsilon \textbf{ must } B \\ \quad \text{implies} & S \textbf{ after } \epsilon \textbf{ must } A \\ \quad \text{implies} & S \textbf{ after } \epsilon \textbf{ must } A \cap out(S) \\ \quad \text{implies} & A \cap out(S) \in \overline{M_\epsilon}(S) \\ \quad \text{implies} & \exists A' \in M \sqsubseteq \overline{M_\epsilon} : \\ & A' \subseteq A \cap out(S) \text{ and } S \textbf{ after } \epsilon \textbf{ must } A' \\ \quad \text{implies} & t' = \Sigma\{a; \mathbf{stop} \mid a \in A'\} \in \Pi \\ & \text{and } out(t') = A' \subseteq A \text{ and } v(t') = \mathbf{fail} \end{array}$$

$a \cdot \sigma$: To prove:

$$\begin{array}{l} \forall S \in \mathcal{LTS}, \forall A \subseteq L : \\ \quad \exists t \in \Pi_R(S) : v(\, t \textbf{ after } a \cdot \sigma \,) = \mathbf{fail} \text{ and } out(\, t \textbf{ after } a \cdot \sigma \,) \subseteq A \\ \text{implies} \\ \quad \exists t' \in \Pi : v(\, t' \textbf{ after } a \cdot \sigma \,) = \mathbf{fail} \text{ and } out(\, t' \textbf{ after } a \cdot \sigma \,) \subseteq A \end{array}$$

$$\begin{array}{ll} \text{Let } t \in \Pi_R(S), & \\ \quad \text{then} & \exists t_a \in \Pi_R(\, \textbf{choice } S \textbf{ after } a \,) : t \xrightarrow{a} t_a \stackrel{\sigma}{\Longrightarrow} t_a \textbf{ after } \sigma \\ & \text{and } v(\, t_a \textbf{ after } \sigma \,) = \mathbf{fail} \text{ and } out(\, t_a \textbf{ after } \sigma \,) \subseteq A \\ \quad \text{implies} & (* \text{ induction hypothesis } *) \\ & \exists t'_a \in \Pi(\, \textbf{choice } S \textbf{ after } a \,) : \\ & v(\, t'_a \textbf{ after } \sigma \,) = \mathbf{fail} \text{ and } out(\, t_a \textbf{ after } \sigma \,) \subseteq A \end{array}$$

implies $(*\ a \in out(t) \subseteq out(S)\ *)$
if $\overline{M_\epsilon}(S) = \emptyset$ then
$t' = \Sigma\{b; t_b \mid b \in out(S)\} \xrightarrow{a} t_a$ and $v(t') = \mathbf{pass}$
and $v(\, t'$ **after** $a{\cdot}\sigma\,) = \mathbf{fail}$ and $out(\, t'$ **after** $a{\cdot}\sigma\,) \subseteq A$
and if $\overline{M_\epsilon}(S) \neq \emptyset$ then
$\exists A' \in M \sqsubseteq \overline{M_\epsilon}(S) : a \in A'$ and
$t' = \Sigma\{b; t_b \mid b \in A'\} \xrightarrow{a} t_a$ and $v(t') = \mathbf{fail}$
and $v(\, t'$ **after** $a{\cdot}\sigma\,) = \mathbf{fail}$ and $out(\, t'$ **after** $a{\cdot}\sigma\,) \subseteq A$
implies $\exists t' \in \Pi :$
$v(\, t'$ **after** $a{\cdot}\sigma\,) = \mathbf{fail}$ and $out(\, t'$ **after** $a{\cdot}\sigma\,) \subseteq A$

$\supseteq$: Directly from $\Pi \subseteq \Pi_R(S)$.

□

Proposition 4.14
Let $Can(S) =_{def} \Sigma\{\ \mathbf{i}; \Sigma\{\ a; Can(\,$ **choice** S **after** $a\,) \mid a \in A\ \}\ \mid A \in M_S\ \}$,

with $\begin{cases} M_S \sqsubseteq \overline{M_\epsilon}(S) & \text{if } \overline{M_\epsilon}(S) \neq \emptyset \\ M_S = \{\ \emptyset, out(S)\ \} & \text{if } \overline{M_\epsilon}(S) = \emptyset \end{cases}$

then $Can(S)$ is a canonical tester of S.

Note that there may be many choices for M_S, hence $Can(S)$ is not uniquely determined.

□

Proof (proposition 4.14)

- $\forall S \in \mathcal{LTS}, \forall \sigma \in L^* : S \stackrel{\sigma}{\Longrightarrow}$ iff $Can(S) \stackrel{\sigma}{\Longrightarrow}$,
  by induction on the length of σ:

  ϵ: Evident.

  $a{\cdot}\sigma$: First, $\exists S' : S \stackrel{a}{\Longrightarrow} S'$ iff $Can(S) \stackrel{a}{\Longrightarrow} Can(\,$ **choice** S **after** $a\,)$, using, if $\overline{M_\epsilon}(S) \neq \emptyset$, then $\exists A \in M_S : a \in A$, and if $\overline{M_\epsilon}(S) = \emptyset$, then $out(S) \in M_S$.
  Secondly, if $a \in out(S) = out(Can(S))$, then:
  $S \stackrel{a{\cdot}\sigma}{\Longrightarrow}$
  iff $(*$ proposition 4.3.3, using $a \in traces(S)\ *)$
  choice S **after** $a \stackrel{\sigma}{\Longrightarrow}$
  iff $(*$ induction hypothesis $*)$
  $Can(\,$ **choice** S **after** $a\,) \stackrel{\sigma}{\Longrightarrow}$
  iff $(*\ Can(S) \stackrel{a}{\Longrightarrow} Can(\,$ **choice** S **after** $a\,)\ *)$
  $Can(S) \stackrel{a{\cdot}\sigma}{\Longrightarrow}$

- We prove
  $$\forall S \in \mathcal{LTS} : testreqs_{\mathcal{D}}(\Pi) = testreqs_{\mathcal{N}}(Can(S))$$
  with Π from proposition 4.10. This is equivalent to (propositions 3.22 and 3.26)

  $\forall S \in \mathcal{LTS}, \forall \sigma \in L^*, \forall A \subseteq L :$
  $\exists t \in \Pi : v(\, t$ **after** $\sigma\,) = \mathbf{fail}$ and $out(\, t$ **after** $\sigma\,) \subseteq A$
  iff $Can(S)$ **after** σ **must** L and $\exists C'\ (\ Can(S) \stackrel{\sigma}{\Longrightarrow} C'$ and $out(C') \subseteq A\ )$

by induction on the length of σ:

ϵ: To prove:

$$\begin{array}{l}\forall S \in \mathcal{LTS}, \forall A \subseteq L : \\ \quad \exists t \in \Pi : v(t) = \mathbf{fail} \;\text{ and }\; out(t) \subseteq A \\ \text{iff } Can(S) \;\mathbf{after}\; \epsilon \;\mathbf{must}\; L \text{ and } \exists C' \;(\; Can(S) \stackrel{\epsilon}{\Longrightarrow} C' \;\text{ and }\; out(C') \subseteq A \;)\end{array}$$

Let M_2 denote M_S for Π, and let M_{Can} denote M_S for $Can(S)$, then

$$\begin{array}{lll}\textit{only if}: & & \exists t \in \Pi : v(t) = \mathbf{fail} \;\text{ and }\; out(t) \subseteq A \\ & \text{implies} & (* \;\text{ definition } \Pi \text{ in proposition 4.10 } *) \\ & & \exists B \in M_2 \sqsubseteq \overline{M_\epsilon}(S) : B \subseteq A \\ & \text{implies} & (* \;\text{ definition 4.1.2: } *) \\ & & (* \;\; rcl(M_{Can}) = rcl(\overline{M_\epsilon}(S)) \text{ implies } *) \\ & & (* \;\; \forall B \in \overline{M_\epsilon}(S), \exists B' \in M_{Can} : B' \subseteq B \;\; *) \\ & & \exists B \in \overline{M_\epsilon}(S), \exists B' \in M_{Can} \sqsubseteq \overline{M_\epsilon}(S) : B' \subseteq B \subseteq A \\ & \text{implies} & (* \;\; \emptyset \notin M_{Can} \neq \emptyset \text{ and definition } Can(S) \;\; *) \\ & & Can(S) \;\mathbf{after}\; \epsilon \;\mathbf{must}\; L \;\text{ and} \\ & & \exists B' \in M_{Can} : B' \subseteq A \;\text{ and} \\ & & Can(S) \stackrel{\epsilon}{\Longrightarrow} \Sigma\{a; Can(\,\mathbf{choice}\; S \;\mathbf{after}\; a\,) \mid a \in B'\} \\ & \text{implies} & Can(S) \;\mathbf{after}\; \epsilon \;\mathbf{must}\; L \;\text{ and} \\ & & \exists C' \;(\; Can(S) \stackrel{\epsilon}{\Longrightarrow} C' \;\text{ and }\; out(C') \subseteq A \;)\end{array}$$

$$\begin{array}{lll}\textit{if}: & & Can(S) \;\mathbf{after}\; \epsilon \;\mathbf{must}\; L \;\text{ and} \\ & & \exists C' \;(\; Can(S) \stackrel{\epsilon}{\Longrightarrow} C' \;\text{ and }\; out(C') \subseteq A \;) \\ & \text{implies} & (\; \exists B \in M_{Can} \sqsubseteq \overline{M_\epsilon}(S) \neq \emptyset \;\text{ and }\; B \subseteq A \;\text{ and} \\ & & Can(S) \stackrel{\epsilon}{\Longrightarrow} \Sigma\{a; Can(\,\mathbf{choice}\; S \;\mathbf{after}\; a\,) \mid a \in B\} \;) \\ & & \text{or } (\; Can(S) \stackrel{\epsilon}{\Longrightarrow} Can(S) \;\text{ and }\; out(Can(S)) = out(S) \subseteq A \;) \\ & \text{implies} & \exists B \in \overline{M_\epsilon}(S), \exists B' \in M_2 \sqsubseteq \overline{M_\epsilon}(S) \neq \emptyset : B' \subseteq B \subseteq A \\ & \text{implies} & \exists B' \in M_2 \neq \emptyset : B' \subseteq A \\ & \text{implies} & \exists B' \in M_2, \exists t = \Sigma\{b; \mathbf{stop} \mid b \in B'\} : \\ & & v(t) = \mathbf{fail} \;\text{ and }\; out(t) = B' \subseteq A \;\text{ and }\; v(\mathbf{stop}) = \mathbf{pass} \\ & \text{implies} & \exists t \in \Pi : v(t) = \mathbf{fail} \;\text{ and }\; out(t) \subseteq A\end{array}$$

$a \cdot \sigma$: To prove:

$$\begin{array}{l}\forall S \in \mathcal{LTS}, \forall A \subseteq L : \\ \quad \exists t \in \Pi : v(\,t \;\mathbf{after}\; a \cdot \sigma\,) = \mathbf{fail} \;\text{ and }\; out(\,t \;\mathbf{after}\; a \cdot \sigma\,) \subseteq A \\ \text{iff } \; Can(S) \;\mathbf{after}\; a \cdot \sigma \;\mathbf{must}\; L \;\text{ and} \\ \quad \exists C' \;(\; Can(S) \stackrel{a \cdot \sigma}{\Longrightarrow} C' \;\text{ and }\; out(C') \subseteq A \;)\end{array}$$

$$
\begin{array}{ll}
 & \exists t \in \Pi : v(\, t \textbf{ after } a{\cdot}\sigma \,) = \textbf{fail} \ \text{ and } \ out(\, t \textbf{ after } a{\cdot}\sigma \,) \subseteq A \\
\text{iff} & \exists t_a \in \Pi(\, \textbf{choice } S \textbf{ after } a \,) : t \xrightarrow{a} t_a \stackrel{\sigma}{\Longrightarrow} t_a \textbf{ after } \sigma \ \text{ and} \\
 & v(\, t_a \textbf{ after } \sigma \,) = \textbf{fail} \ \text{ and } \ out(\, t_a \textbf{ after } \sigma \,) \subseteq A \\
\text{iff} & (* \ \text{induction hypothesis} \ *) \\
 & Can(\, \textbf{choice } S \textbf{ after } a \,) \textbf{ after } \sigma \textbf{ must } L \ \text{ and} \\
 & \exists C' \ (\ Can(\, \textbf{choice } S \textbf{ after } a \,) \stackrel{\sigma}{\Longrightarrow} C' \ \text{ and } \ out(C') \subseteq A \ ) \\
\text{iff} & (* \ a \in out(t) \subseteq out(S) = out(Can(S)), \text{ and } \ *) \\
 & (* \ (\ Can(S) \stackrel{a}{\Longrightarrow} Can(\, \textbf{choice } S \textbf{ after } a \,) \ ) \ *) \\
 & Can(S) \textbf{ after } a{\cdot}\sigma \textbf{ must } L \ \text{ and} \\
 & \exists C' \ (\ Can(S) \stackrel{a}{\Longrightarrow} Can(\, \textbf{choice } S \textbf{ after } a \,) \stackrel{\sigma}{\Longrightarrow} C' \ \text{ and } \ out(C') \subseteq A \ )
\end{array}
$$

□

Proposition 4.15

1. If $C_1, C_2 \in \mathcal{LTS}$ are both canonical testers of S, then $C_1 \approx_{te} C_2$.
2. If C_S is a canonical tester of S, and C_{C_S} is a canonical tester of C_S, then $C_{C_S} \approx_{te} S$.

□

Proof (proposition 4.15)
See [Bri87].

□

B.4.2 Section 4.3 (Language Based Test Derivation)

Proposition 4.18
Let $S \in \mathcal{LTS}$, then

1. $$
\begin{array}{ll}
 & \{\ out(S') \mid S \stackrel{\epsilon}{\Longrightarrow} S' \not\xrightarrow{\tau} \ \} \ \cup \ \{out(S)\} \\
\sqsubseteq & \{\ out(S') \mid S \stackrel{\epsilon}{\Longrightarrow} S' \ \} \\
\sqsubseteq & \{\ A \subseteq out(S) \mid \exists S' \ (\ S \stackrel{\epsilon}{\Longrightarrow} S' \ \text{ and } \ out(S') \subseteq A \ ) \ \} \\
= & \overline{C_\epsilon}(S)
\end{array}
$$
2. If $C_1 \sqsubseteq C_2$ then $\Psi(C_1) = \Psi(C_2)$
3. Let $C' \sqsubseteq \overline{C_\epsilon}(S)$, then $\Psi(C') = \overline{M_\epsilon}(S)$
4. Let $M' \sqsubseteq \overline{M_\epsilon}(S)$, then $\Psi(M') = \overline{C_\epsilon}(S)$

□

Proof (proposition 4.18)

1. Let $C_1 = \{out(S') \mid S \stackrel{\epsilon}{\Longrightarrow} S' \not\xrightarrow{\tau} \} \cup \{out(S)\}$, $C_2 = \{out(S') \mid S \stackrel{\epsilon}{\Longrightarrow} S'\}$:
   - $$
\begin{array}{lll}
\overline{C_\epsilon}(S) & = & \{\ out(S) \backslash A \mid \ S \textbf{ after } \epsilon \textbf{ refuses } A \ \} \\
 & = & \{\ A' \subseteq out(S) \mid \ S \textbf{ after } \epsilon \textbf{ refuses } out(S) \backslash A' \ \} \\
 & = & \{\ A' \subseteq out(S) \mid \exists S' \, (\, S \stackrel{\epsilon}{\Longrightarrow} S' \ \text{ and } \ \forall a \in out(S) \backslash A' : S' \stackrel{a}{\not\Longrightarrow} \,) \ \} \\
 & = & \{\ A' \subseteq out(S) \mid \exists S' \ (\ S \stackrel{\epsilon}{\Longrightarrow} S' \ \text{ and } \ out(S') \subseteq A' \ ) \ \} \\
 & \supseteq & \{\ out(S') \mid S \stackrel{\epsilon}{\Longrightarrow} S' \ \} \ = \ C_2 \\
 & \supseteq & \{\ out(S') \mid S \stackrel{\epsilon}{\Longrightarrow} S' \ \text{ and } \ S' \not\xrightarrow{\tau} \ \} \cup \{out(S)\} \ = \ C_1
\end{array}
$$

- $\bigcup \overline{C_\epsilon}(S) = \bigcup C_1 = \bigcup C_2 = out(S)$
- $rcl_\subseteq(C_1) = rcl_\subseteq(C_2) = rcl_\subseteq(\overline{C_\epsilon}(S)) = \overline{C_\epsilon}(S)$:
  $\subseteq$ directly from $C_1 \subseteq C_2 \subseteq \overline{C_\epsilon}(S)$.
  For $\supseteq$, let $A \in \overline{C_\epsilon}(S)$,
  then $\exists S'$ ($S \stackrel{\epsilon}{\Longrightarrow} S'$ and $out(S') \subseteq A \subseteq out(S)$) }
  implies ($*$ S is strongly converging, section 1.4 $*$)
  $\exists S', S''$ ($S \stackrel{\epsilon}{\Longrightarrow} S' \stackrel{\epsilon}{\Longrightarrow} S'' \stackrel{\tau}{\not\longrightarrow}$ and $out(S'') \subseteq out(S') \subseteq A$) }
  implies $\exists A' = out(S') \in C_2 : A' \subseteq A$ and
  $\exists A'' = out(S'') \in C_1 : A'' \subseteq A$
  implies $A \in rcl_\subseteq(C_2)$, and $A \in rcl_\subseteq(C_1)$

2. To prove: $\forall A \subseteq L :$ $A \subseteq \bigcup C_1$ and $\forall A' \in C_1 : A \cap A' \neq \emptyset$
   iff $A \subseteq \bigcup C_2$ and $\forall A'' \in C_2 : A \cap A'' \neq \emptyset$

   $\bigcup C_1 = \bigcup C_2$ from the definition of $C_1 \sqsubseteq C_2$;
   if from $C_1 \subseteq C_2$; for *only if*, let $A'' \in C_2$,
   then ($*$ $rcl_\subseteq(C_1) = rcl_\subseteq(C_2)$ $*$)
   $\exists A''_1 \in C_1 : A''_1 \subseteq A''$
   implies ($*$ $\forall A' \in C_1 : A \cap A' \neq \emptyset$ $*$)
   $A \cap A'' \neq \emptyset$

3. $\subseteq$: Let $A \in \Psi(C') = \Psi(\overline{C_\epsilon}(S))$, and let $S \stackrel{\epsilon}{\Longrightarrow} S'$,
   then $out(S') \in \overline{C_\epsilon}(S)$ and $A \cap out(S') \neq \emptyset$, so $\exists a \in A : S' \stackrel{a}{\Longrightarrow}$,
   hence $A \in \overline{M_\epsilon}(S)$.

   $\supseteq$: Let $A \in \overline{M_\epsilon}(S)$, and let $A' \in C' \subseteq \overline{C_\epsilon}(S)$,
   then $\exists S'$ ($S \stackrel{\epsilon}{\Longrightarrow} S'$ and $out(S') \subseteq A'$).
   For this S', since $A \in \overline{M_\epsilon}(S) : \exists a \in A : S' \stackrel{a}{\Longrightarrow}$, so $A \cap out(S') \neq \emptyset$,
   hence $A \cap A' \neq \emptyset$, hence $A \in \Psi(C')$.

4. $\subseteq$: By contraposition: $A \notin \overline{C_\epsilon}(S)$ implies $A \notin \Psi(M') = \Psi(\overline{M_\epsilon}(S))$.
   From proposition 4.18.1:
   $A \notin \overline{C_\epsilon}(S)$ iff S **after** ϵ **must** $out(S) \backslash A$ iff $out(S) \backslash A \in \overline{M_\epsilon}(S)$.
   Moreover, $(out(S) \backslash A) \cap A = \emptyset$, hence $A \notin \Psi(\overline{M_\epsilon}(S))$.

   $\supseteq$: Let $A \in \overline{C_\epsilon}(S)$, and let $A' \in M' \subseteq \overline{M_\epsilon}(S)$,
   then $\exists S'$ ($S \stackrel{\epsilon}{\Longrightarrow} S'$ and $out(S') \subseteq A$).
   For this S', since $A' \in \overline{M_\epsilon}(S)$, $\exists a \in A' : S' \stackrel{a}{\Longrightarrow}$,
   hence $out(S') \cap A' \neq \emptyset$, hence $A \cap A' \neq \emptyset$, hence $A \in \Psi(M')$.

□

Proposition 4.20

1. If $S \in \mathcal{LTS}$ is image-finite, then
   - there is a finite $C' \sqsubseteq \overline{C_\epsilon}(S)$;
   - for any σ: **choice** S **after** σ is image-finite.
2. If $C' \sqsubseteq \overline{C_\epsilon}(S)$ is finite, then

- $\Psi_o(C') \sqsubseteq \Psi(C')$;
- $min_{\subseteq}(C') \cup \{out(S)\} \ \sqsubseteq \ \overline{C_\epsilon}(S)$;
- $min_{\subseteq}(\Psi_o(C')) \cup \{out(S) \mid \emptyset \notin C'\} \ \sqsubseteq \ \Psi(\overline{C_\epsilon}(S))$.

3. The language $\mathcal{BEX}$ restricted to (4.6) is image-finite.

□

Proof (lemma 4.20)

1. - S is image-finite iff $\forall \sigma \in L^* : S \,\mathbf{after}\, \sigma$ is finite, (definition 1.11.6),
   implies $\{ S' \mid S \stackrel{\epsilon}{\Longrightarrow} S' \}$ is finite,
   implies $C' = \{ out(S') \mid S \stackrel{\epsilon}{\Longrightarrow} S' \}$ is finite (proposition 4.18.1).
   - Image-finiteness of $\mathbf{choice}\ S\ \mathbf{after}\ \sigma$ follows directly from definition 4.2 and statement (1) in the proof of proposition 4.3.
2. - $\Psi_o(C) \subseteq \Psi(C)$: Let $A = \{a_1, \ldots, a_n\} \in \Psi_o(C)$, then $\forall i : a_i \in A_i$, hence $\forall i : A \cap A_i \neq \emptyset$, and $A \subseteq \bigcup\{A_1, \ldots, A_n\} = \bigcup C$, so $A \in \Psi(C)$.

   $rcl_{\subseteq}(\Psi_o(C)) = rcl_{\subseteq}(\Psi(C))$: $\subseteq$ directly from $\Psi_o(C) \subseteq \Psi(C)$.

   For $\supseteq$: $A \in rcl_{\subseteq}(\Psi(C))$
   implies $\exists A' \in \Psi(C) : A' \subseteq A$
   implies $\exists A', \forall i : A' \cap A_i \neq \emptyset$ and $A' \subseteq A$
   implies $\exists A', \forall i, \exists a_i \in A_i : a_i \in A'$ and $A' \subseteq A$
   implies $\exists A', \exists \{a_1, \ldots, a_n\} \in \Psi_o(C) : \{a_1, \ldots, a_n\} \subseteq A' \subseteq A$
   implies $A \in rcl_{\subseteq}(\Psi_o(C))$

   $\bigcup \Psi_o(C) = \bigcup \Psi(C)$: If $\emptyset \in C$ then $\bigcup \Psi_o(C) = \bigcup \Psi(C) = \emptyset$, else:
   $a \in \bigcup \Psi(C)$
   iff $a \in \bigcup C$
   iff $\exists A_i \in C : a \in A_i$
   iff $\exists \{a_1, \ldots, a_n\} \in \Psi_o(C) : a \in \{a_1, \ldots, a_n\}$
   iff $a \in \bigcup \Psi_o(C)$

   - Let $C'' = min_{\subseteq}(C') \cup \{out(S)\}$:

   $C'' \subseteq \overline{C_\epsilon}(S)$: $A \in C''$ implies either $A \in min_{\subseteq}(C') \subseteq C' \subseteq \overline{C_\epsilon}(S)$, or $A = out(S) \in \overline{C_\epsilon}(S)$.

   $\bigcup C'' = \bigcup \overline{C_\epsilon}(S)$: Both are equal to $out(S)$.

   $rcl_{\subseteq}(C'') = rcl_{\subseteq}(\overline{C_\epsilon}(S))$: $\overline{C_\epsilon}(S) \neq \emptyset$ implies $C' \neq \emptyset$ implies
   $(*\ C'$ is finite: existence of minimal elements $*)$ $min(C') \neq \emptyset$;
   moreover, $\forall A \in C' : A \subseteq out(S)$,
   hence $min_{\subseteq}(C'') = min_{\subseteq}(min_{\subseteq}(C') \cup \{out(S)\}) = min_{\subseteq}(C')$,
   implying $(*\ C'$ and C'' are finite: proposition A.2 $*)$:
   $rcl_{\subseteq}(C'') = rcl_{\subseteq}(C') = (*\ C' \sqsubseteq \overline{C_\epsilon}(S)\ *) = rcl_{\subseteq}(\overline{C_\epsilon}(S))$.

   - Let $M = min_{\subseteq}(\Psi_o(C')) \cup \{out(S) \mid \emptyset \notin C'\}$, then, if $\emptyset \in C'$, $M = \emptyset = \Psi(\overline{C_\epsilon}(S))$.
   If $\emptyset \notin C'$, the proof is analogous to the previous item:

$M \subseteq \Psi(\overline{C_\epsilon}(S))$: $A \in M$ implies either $A \in min_\subseteq(\Psi_o(C')) \subseteq \Psi_o(C') \subseteq \Psi(C') = \Psi(\overline{C_\epsilon}(S))$, or $A = out(S) \in \Psi\overline{C_\epsilon}(S))$.

$\bigcup M = \bigcup \Psi(\overline{C_\epsilon}(S))$: Both are equal to $out(S)$.

$rcl_\subseteq(M) = rcl_\subseteq(\Psi(\overline{C_\epsilon}(S))$: $\emptyset \notin C'$ implies $\Psi_o(C') \neq \emptyset$ implies
($*$ $\Psi_o(C')$ is subset of the set of finite subsets of $out(S)$, $*$)
($*$ which is well-founded $*$) $min(\Psi_o(C')) \neq \emptyset$;
moreover, $\forall A \in \Psi_o(C') : A \subseteq out(S)$,
hence $min_\subseteq(M) = min_\subseteq(min_\subseteq(\Psi_o(C')) \cup \{out(S)\}) = min_\subseteq(\Psi_o(C'))$,
implying ($*$ well-foundedness: proposition A.2 $*$) :
$rcl_\subseteq(M) = rcl_\subseteq(\Psi_o(C')) = (* \quad \Psi_o(C') \sqsubseteq \Psi(C') \quad *) = rcl_\subseteq(\Psi(C')) = rcl_\subseteq(\Psi(\overline{C_\epsilon}(S)))$.

3. By induction on the structure of $B \in \mathcal{BEX}$:
   - $\mathbf{stop}\ \mathbf{after}\ \epsilon = \{\mathbf{stop}\},\ \mathbf{stop}\ \mathbf{after}\ a{\cdot}\sigma = \emptyset$, hence always finite.
   - $a; B\ \mathbf{after}\ \epsilon = a; B,\ a; B\ \mathbf{after}\ a{\cdot}\sigma = B\ \mathbf{after}\ \sigma\ ,\ a; B\ \mathbf{after}\ b{\cdot}\sigma = \emptyset\ (b \neq a)$, hence always finite, if $B\ \mathbf{after}\ \sigma$ is finite.
   - $\mathbf{i}; B\ \mathbf{after}\ \epsilon = B\ \mathbf{after}\ \epsilon \cup \{\mathbf{i}; B\},\ \mathbf{i}; B\ \mathbf{after}\ a{\cdot}\sigma = B\ \mathbf{after}\ a{\cdot}\sigma$, hence always finite, if $B\ \mathbf{after}\ a{\cdot}\sigma$ is finite.
   - $B_1 \Box B_2\ \mathbf{after}\ \epsilon = \{B_1 \Box B_2\} \cup (\ B_1\ \mathbf{after}\ \epsilon\ ) \backslash \{B_1\} \cup (\ B_2\ \mathbf{after}\ \epsilon\ ) \backslash \{B_2\}$,
     $B_1 \Box B_2\ \mathbf{after}\ a{\cdot}\sigma = (\ B_1\ \mathbf{after}\ a{\cdot}\sigma\ ) \backslash \{B_1\} \cup (\ B_2\ \mathbf{after}\ a{\cdot}\sigma\ ) \backslash \{B_2\}$, hence always finite, if $B_1\ \mathbf{after}\ \sigma$ and $B_2\ \mathbf{after}\ \sigma$ are finite for all σ.

□

Proposition 4.24
Let $B \in \mathcal{BEX}$, restricted to (4.6), and let $out_\mathcal{B}(B)$, $st_\mathcal{B}(B)$, $\mathcal{C}(B)$, and $\mathbf{choice}_\mathcal{B}\ B\ \mathbf{after}\ g$ be compositionally defined in table 4.1, then

- $out_\mathcal{B}(B) = out(\ell ts(B))$
- $st_\mathcal{B}(B)$ iff $\ell ts(B)$ is stable
- $\mathcal{C}(B) \sqsubseteq \overline{C_\epsilon}(\ell ts(B))$
- $\mathbf{choice}_\mathcal{B}\ B\ \mathbf{after}\ g \approx_{te} \mathbf{choice}\ \ell ts(B)\ \mathbf{after}\ g$

□

Proof (proposition 4.24)

1. $out(\ell ts(\mathbf{stop}))$
   $= (*$ definition $out(S)$ in 1.11.1 $*)$
   $\{\ a \in L \mid \ell ts(\mathbf{stop}) \stackrel{a}{\Longrightarrow}\ \}$
   $= (*$ $\mathbf{stop}$ as the initial state in $\ell ts(\mathbf{stop}) = \langle \mathcal{BEX}, L, T_{\mathcal{BEX}}, \mathbf{stop} \rangle$ $*)$
   $\{\ a \in L \mid \mathbf{stop} \stackrel{a}{\Longrightarrow}\ \}$
   $= (*$ definition 1.7 $*)$
   $\emptyset$
   $= (*$ table 4.1, $\mathbf{stop}$ as an element of $\mathcal{BEX}$ $*)$
   $out_\mathcal{B}(\mathbf{stop})$

2. With analogous argumentations:
   $out(\ell ts(a;B_1)) = \{\ a \in L \mid a;B_1 \stackrel{a}{\Longrightarrow} \} = \{a\}$
3. $out(\ell ts(\mathbf{i};B_1)) = \{\ a \in L \mid \mathbf{i};B_1 \stackrel{a}{\Longrightarrow} \} = \{\ a \in L \mid \mathbf{i};B_1 \stackrel{\tau}{\longrightarrow} B_1 \stackrel{a}{\Longrightarrow} \}$
   $= \{\ a \in L \mid B_1 \stackrel{a}{\Longrightarrow} \} = out_{\mathcal{B}}(B_1)$
4. $out(\ell ts(B_1 \Box B_2)) = \{\ a \in L \mid B_1 \Box B_2 \stackrel{a}{\Longrightarrow}\ \} = \{\ a \in L \mid B_1 \stackrel{a}{\Longrightarrow}$ or $B_2 \stackrel{a}{\Longrightarrow}\ \}$
   $= \{\ a \in L \mid B_1 \stackrel{a}{\Longrightarrow} \} \cup \{\ a \in L \mid B_2 \stackrel{a}{\Longrightarrow} \} = out_{\mathcal{B}}(B_1) \cup out_{\mathcal{B}}(B_2)$
5. $st(\ell ts(\mathbf{stop}))$
   iff (∗ definition *stable* in 1.11.8 ∗)
   $\ell ts(\mathbf{stop}) \stackrel{\tau}{\not\longrightarrow}$
   iff (∗ **stop** as the initial state in $\ell ts(\mathbf{stop}) = \langle \mathcal{BEX}, L, T_{\mathcal{BEX}}, \mathbf{stop} \rangle$ ∗)
   $\mathbf{stop} \stackrel{\tau}{\not\longrightarrow}$
   iff (∗ definition 1.7 ∗)
   true
   = (∗ table 4.1, **stop** as an element of $\mathcal{BEX}$ ∗)
   $st_{\mathcal{B}}(\mathbf{stop})$
6. With analogous argumentations:
   $st(\ell ts(a;B_1))$ iff $a;B_1 \stackrel{\tau}{\not\longrightarrow}$ always holds.
7. $st(\ell ts(\mathbf{i};B_1))$ iff $\mathbf{i};B_1 \stackrel{\tau}{\not\longrightarrow}$ never holds.
8. $st(\ell ts(B_1 \Box B_2))$ iff $B_1 \Box B_2 \stackrel{\tau}{\not\longrightarrow}$
   iff $B_1 \stackrel{\tau}{\not\longrightarrow}$ and $B_2 \stackrel{\tau}{\not\longrightarrow}$ iff $st_{\mathcal{B}}(B_1)$ and $st_{\mathcal{B}}(B_2)$
9. $\overline{C_\epsilon}(\ell ts(\mathbf{stop})) \sqsupseteq$ (∗ proposition 4.18.1 ∗)
   $\{\ out(S') \mid \mathbf{stop} \stackrel{\epsilon}{\Longrightarrow} S'\ \} = \{out(\mathbf{stop})\} = \{\emptyset\}$
10. $\overline{C_\epsilon}(\ell ts(a;B_1)) \sqsupseteq \{\ out(S') \mid a;B_1 \stackrel{\epsilon}{\Longrightarrow} S'\ \} = \{out(a;B_1)\} = \{\{a\}\}$
11. To prove: $\mathcal{C}(\mathbf{i};B_1) \sqsubseteq \overline{C_\epsilon}(\ell ts(\mathbf{i};B_1))$

    First: $\overline{C_\epsilon}(\mathbf{i};B_1)$
    = $\{\ A \subseteq out(\mathbf{i};B_1) \mid \exists S'\ (\ \mathbf{i};B_1 \stackrel{\epsilon}{\Longrightarrow} S'$ and $out(S') \subseteq A\ )\ \}$
    = $\{\ A \subseteq out(B_1) \mid \exists S'\ (\ B_1 \stackrel{\epsilon}{\Longrightarrow} S'$ and $out(S') \subseteq A\ )\ \}$
    = $\overline{C_\epsilon}(B_1)$

    Secondly, we may assume: $\mathcal{C}(B_1) \sqsubseteq \overline{C_\epsilon}(B_1)$.

    Together: $\mathcal{C}(\mathbf{i};B_1) =$ (∗ table 4.1 ∗) $= \mathcal{C}(B_1) \sqsubseteq \overline{C_\epsilon}(B_1) = \overline{C_\epsilon}(\mathbf{i};B_1)$.
12. To prove: $\mathcal{C}(B_1 \Box B_2) \sqsubseteq \overline{C_\epsilon}(\ell ts(B_1 \Box B_2))$

    where $\mathcal{C}(B_1 \Box B_2) = \{out(B_1 \Box B_2)\} \cup$ if $\not st(B_1)$ then $\mathcal{C}(B_1) \cup$ if $\not st(B_2)$ then $\mathcal{C}(B_2)$.
    According to the definition of $\sqsubseteq$ in definition 4.1.2:

    $\bigcup \mathcal{C}(B_1 \Box B_2) = \bigcup \overline{C_\epsilon}(B_1 \Box B_2)$: Both are equal to $out(B_1 \Box B_2)$.

    $\mathcal{C}(B_1 \Box B_2) \subseteq \overline{C_\epsilon}(B_1 \Box B_2)$: Let $A \in \mathcal{C}(B_1 \Box B_2)$, then (1), (2), or (3):
    (1): $A = out(B_1 \Box B_2)$
    implies $A \in \overline{C_\epsilon}(B_1 \Box B_2)$

(2): $A \in \mathcal{C}(B_1)$
implies $\not\delta t(B_1)$ and $A \in \overline{C_\epsilon}(B_1)$
implies $\exists S'$ ($B_1 \stackrel{\epsilon}{\Longrightarrow} S'$ and $out(S') \subseteq A$)
implies
either $\exists S'$ ($B_1 \stackrel{\tau}{\longrightarrow} \stackrel{\epsilon}{\Longrightarrow} S'$ and $out(S') \subseteq A$)
implies $B_1 \Box B_1 \stackrel{\tau}{\longrightarrow} \stackrel{\epsilon}{\Longrightarrow} S'$ and $out(S') \subseteq A$)
implies $A \in \overline{C_\epsilon}(B_1 \Box B_2)$
or $B_1 \stackrel{\epsilon}{\Longrightarrow} B_1$ and $out(B_1) \subseteq A$
implies $\exists B_1'$ ($B_1 \stackrel{\tau}{\longrightarrow} B_1'$ and $out(B_1') \subseteq out(B_1) \subseteq A$)
implies $\exists B_1'$ ($B_1 \Box B_2 \stackrel{\tau}{\longrightarrow} B_1'$ and $out(B_1') \subseteq A$)
implies $A \in \overline{C_\epsilon}(B_1 \Box B_2)$
(3): $A \in \mathcal{C}(B_2)$: analogous to (2), exchanging B_1 and B_2 .

$rcl_\subseteq(\mathcal{C}(B_1 \Box B_2)) = rcl_\subseteq(\overline{C_\epsilon}(B_1 \Box B_2))$: $\subseteq$ directly from $\mathcal{C}(B_1 \Box B_2) \subseteq \overline{C_\epsilon}(B_1 \Box B_2)$.
For $\supseteq$, let $A \in rcl_\subseteq(\overline{C_\epsilon}(B_1 \Box B_2)) = \overline{C_\epsilon}(B_1 \Box B_2))$,
then $\exists S'$ ($B_1 \Box B_2 \stackrel{\epsilon}{\Longrightarrow} S'$ and $out(S') \subseteq A$):

$S' = B_1 \Box B_2$: $A = out(S') = out(B_1 \Box B_2) \in rcl_\subseteq(\mathcal{C}(B_1 \Box B_2))$

$S' = B_1'$: $B_1 \Box B_2 \stackrel{\tau}{\longrightarrow} \stackrel{\epsilon}{\Longrightarrow} B_1'$ and $out(B_1') \subseteq A$
implies $A \in \overline{C_\epsilon}(B_1)$ and $\not\delta t(B_1)$
implies $(* \ \mathcal{C}(B_1) \sqsubseteq \overline{C_\epsilon}(B_1) \ *)$
$\exists A' \in \mathcal{C}(B_1) : A' \subseteq A$ and $\not\delta t(B_1)$
implies $\exists A' \in \mathcal{C}(B_1 \Box B_2) : A' \subseteq A$
implies $A \in rcl_\subseteq(\mathcal{C}(B_1 \Box B_2))$

$S' = B_2'$: Analogous to the previous item, exchanging B_1 and B_2.

13. **choice** $\ell ts(a; B_1)$ **after** g
$\approx_{te}$ $(*$ proposition 4.3.1 $*)$
$\Sigma\{\mathbf{i}; S' \mid a; B_1 \stackrel{g}{\Longrightarrow} S'\}$
$=$ $(*$ definition $\stackrel{g}{\Longrightarrow}$ in table 1.1 $*)$
$\Sigma\{\mathbf{i}; B_1 \mid a = g\}$
$= \begin{cases} \mathbf{i}; B_1 & \text{if } a = g \\ \mathbf{stop} & \text{if } a \neq g \end{cases}$
$=$ $(*$ table 4.1 $*)$
$\mathbf{choice}_{\mathcal{B}}\ a; B$ **after** g

14. With analogous argumentations:
choice $\ell ts(\mathbf{stop})$ **after** $g \approx_{te} \Sigma\{\mathbf{i}; S' \mid \mathbf{stop} \stackrel{g}{\Longrightarrow} S'\} = \Sigma\emptyset = \mathbf{stop}$

15. **choice** $\ell ts(\mathbf{i}; B_1)$ **after** $g \approx_{te} \Sigma\{\mathbf{i}; S' \mid \mathbf{i}; B_1 \stackrel{g}{\Longrightarrow} S'\} = \Sigma\{\mathbf{i}; S' \mid B_1 \stackrel{g}{\Longrightarrow} S'\} \approx_{te}$ $\mathbf{choice}_{\mathcal{B}}\ B_1$ **after** g

16. **choice** $\ell ts(B_1 \Box B_2)$ **after** $g \approx_{te} \Sigma\{\mathbf{i}; S' \mid B_1 \Box B_2 \stackrel{g}{\Longrightarrow} S'\}$
$= \Sigma\{\mathbf{i}; S' \mid B_1 \stackrel{g}{\Longrightarrow} S' \text{ or } B_2 \stackrel{g}{\Longrightarrow} S'\}$
$\equiv \Sigma\{\mathbf{i}; S' \mid B_1 \stackrel{g}{\Longrightarrow} S'\} \ \Box \ \Sigma\{\mathbf{i}; S' \mid B_2 \stackrel{g}{\Longrightarrow} S'\}$

$= \mathbf{choice}_{\mathcal{B}}\ B_1\ \mathbf{after}\ g\ \Box\ \mathbf{choice}_{\mathcal{B}}\ B_2\ \mathbf{after}\ g$

□

B.4.3 Section 4.4 (Test Derivation with Values)

Proposition 4.28
Any $B \in \mathcal{BEX}_v^c$ is image-finite.

□

Proof (proposition 4.28)
By induction on the structure of $B \in \mathcal{BEX}_v$:

- $\mathbf{stop}\ \mathbf{after}\ \epsilon = \{\mathbf{stop}\};\ \mathbf{stop}\ \mathbf{after}\ a{\cdot}\sigma = \emptyset$; hence always finite.
- $g?x : p; B\ \mathbf{after}\ \epsilon = \Sigma\{\langle g, v\rangle; B[v/x] \mid p[v/x]\}\ \mathbf{after}\ \epsilon = \Sigma\{\langle g, v\rangle; B[v/x] \mid p[v/x]\}$.
  $g?x : p; B\ \mathbf{after}\ \langle h, v\rangle{\cdot}\sigma = \Sigma\{\langle g, v\rangle; B[v/x] \mid p[v/x]\}\ \mathbf{after}\ \langle h, v\rangle{\cdot}\sigma =$ if $g = h$ and $p[v/x]$ then $B[v/x]\ \mathbf{after}\ \sigma$ else $\emptyset$.
  Hence it is always finite, if $B[v/x]\ \mathbf{after}\ \sigma$ is finite. Since $B[v/x]$ is closed, this follows from the induction hypothesis.
- If p then $\mathbf{i} : p; B\ \mathbf{after}\ \epsilon = B\ \mathbf{after}\ \epsilon \cup \{\mathbf{i}; B\}$, and $\mathbf{i} : p; B\ \mathbf{after}\ \langle h, v\rangle{\cdot}\sigma = \mathbf{i}; B\ \mathbf{after}\ \langle h, v\rangle{\cdot}\sigma = B\ \mathbf{after}\ \sigma$.
  If not p then $\mathbf{i} : p; B\ \mathbf{after}\ \epsilon = \{\mathbf{stop}\}$, and $\mathbf{i} : p; B\ \mathbf{after}\ \langle h, v\rangle{\cdot}\sigma = \emptyset$.
  Hence it is always finite, if $B\ \mathbf{after}\ \langle h, v\rangle{\cdot}\sigma$ is finite.
- $B_1 \Box B_2\ \mathbf{after}\ \epsilon = \{B_1 \Box B_2\} \cup (\ B_1\ \mathbf{after}\ \epsilon\ ) \backslash \{B_1\} \cup (\ B_2\ \mathbf{after}\ \epsilon\ ) \backslash \{B_2\}$,
  $B_1 \Box B_2\ \mathbf{after}\ a{\cdot}\sigma = (\ B_1\ \mathbf{after}\ a{\cdot}\sigma\ ) \backslash \{B_1\} \cup (\ B_2\ \mathbf{after}\ a{\cdot}\sigma\ ) \backslash \{B_2\}$, hence always finite, if $B_1\ \mathbf{after}\ \sigma$ and $B_2\ \mathbf{after}\ \sigma$ are finite for all σ.

□

Lemma 4.30
Let $C \in \mathcal{P}(\mathcal{P}(L))$, $D \in \mathcal{P}(\mathcal{P}(\mathcal{P}(L)))$, such that there is a bijection $\delta : C \to D$, with for all $A \in C : \bigcup \delta(A) = A$, then

$$\Psi(C) \quad = \quad \bigcup \{\ \Psi(E) \mid E \in \Psi(D)\ \}$$

□

Proof (lemma 4.30)

$\subseteq$: Let $\alpha \in \Psi(C)$,

then:	$\alpha \in \bigcup C$	and $\forall A \in C : \alpha \cap A \neq \emptyset$
implies	$\alpha \in \bigcup C$	and $\forall A \in C : \alpha \cap \bigcup \delta(A) \neq \emptyset$
implies	$\alpha \in \bigcup C$	and $\forall A \in C : \exists D'' \in \delta(A) : \alpha \cap D'' \neq \emptyset$
implies	$\alpha \in \bigcup C$	and $\forall D' \in D : \exists D'' \in D' : \alpha \cap D'' \neq \emptyset$

Let $F = \{\ D'' \mid \exists D' \in D : D'' \in D'$ and $D'' \cap \alpha \neq \emptyset\ \}$, then

then:	$\forall D'' \in F : \alpha \cap F \neq \emptyset$	(1)
and	$a \in \alpha$	
implies	$\exists A \in C : a \in A$	
implies	$\exists D' \in D : a \in \bigcup D'$	
implies	$\exists D' \in D, \exists D'' \in D' : a \in D''$	
implies	$\exists D'' \in F : a \in D''$	
implies	$a \in \bigcup F$	(2)
thus	from (1) and (2): $\alpha \in \Psi(F)$	(3)

moreover:	$F \subseteq \bigcup D$ and $\forall D' \in D : \exists D'' \in D' : \alpha \cap D'' \neq \emptyset$	
implies	$F \subseteq \bigcup D$ and $\forall D' \in D : F \cap D' \neq \emptyset$	
implies	$F \in \Psi(D)$	(4)

Concluding from (3) and (4): $\alpha \in \bigcup\{\Psi(E) \mid E \in \Psi(D)\}$.

$\supseteq$: Let $\alpha \in \bigcup\{\Psi(E) \mid E \in \Psi(D)\}$,

then:	$\exists E \in \Psi(D) : \alpha \in \Psi(E)$.
implies	$\exists E : (\ E \subseteq \bigcup D$ and $\forall D' \in D : E \cap D' \neq \emptyset\ )$ and
	$(\ \alpha \subseteq \bigcup E$ and $\forall E' \in E : \alpha \cap E' \neq \emptyset\ )$
implies	$\exists E : \alpha \subseteq \bigcup E$ and $E \subseteq \bigcup D$ and
	$\forall D' \in D : \exists D'' \in D' : D'' \in E$ and $\forall E' \in E : \alpha \cap E' \neq \emptyset$
implies	$\alpha \subseteq \bigcup\bigcup D$ and $\forall D' \in D : \exists D'' \in D' : \alpha \cap D'' \neq \emptyset$
implies	$\alpha \subseteq \bigcup C$ and $\forall D' \in D : \alpha \cap \bigcup D' \neq \emptyset$
implies	$\alpha \subseteq \bigcup C$ and $\forall A \in C : \alpha \cap A \neq \emptyset$
implies	$\alpha \in \Psi(C)$

□

B.5 Chapter 5 (Asynchronous Testing)

B.5.1 Section 5.4 (Queue Equivalence)

Proposition 5.11

1. Any queue context Q can always do any sequence of input actions: $\forall \sigma_i \in L_I^* : Q \stackrel{\sigma_i}{\Longrightarrow}$
2. For a queue context Q, $\sigma \in L^*$, $A \subseteq L$, and $A \cap L_I \neq \emptyset$, Q **after** σ **must** A holds.

□

Proof (proposition 5.11)

1. From axiom $A1_Q$ it follows that a queue context can always do an input action. Together with the fact that for any derivation $Q \stackrel{\sigma}{\Longrightarrow} Q'$, with $\sigma \in L^*$ and Q a queue context, Q' is a queue context too, as follows immediately from $A1_Q$, $A2_Q$, $I1_Q$, $I2_Q$ and $I3_Q$, it follows that any queue context can always do any sequence of input actions.
2. A direct consequence of proposition 5.11.1, again using the fact that a derivation of a queue context results in a queue context.

□

Proposition 5.12
If $Q \stackrel{\sigma \cdot x}{\Longrightarrow}$ then Q **after** σ **refuses** $L_U \backslash \{x\}$.

□

Proof (proposition 5.12)
The proof uses the fact that if the first action in the output queue of a queue context Q is x, then Q can do an x-action, but no other output action:

$$\text{if } Q \stackrel{x}{\longrightarrow} \text{ then } \forall y \in L_U, y \neq x : Q \stackrel{y}{\Longrightarrow\!\!\!\!\!\!/}$$

which is seen as follows: if $Q \stackrel{x}{\longrightarrow}$ then this must be a consequence of $A2_Q$, and Q must have the form $[_{x \cdot \sigma_u \ll} S_{\ll \sigma_i}]$. It follows that $[_{x \cdot \sigma_u \ll} S_{\ll \sigma_i}] \stackrel{y}{\not\longrightarrow}$ if $y \in L_U$ and $y \neq x$. But since $A2_Q$ is the only axiom or inference rule that can remove x from the head of the output queue, it follows that Q must first do x before it can do any output action $y \neq x$, and thus $Q \stackrel{y}{\Longrightarrow\!\!\!\!\!\!/}$.

Now: $Q \stackrel{\sigma \cdot x}{\Longrightarrow}$
implies $\exists Q' : Q \stackrel{\sigma}{\Longrightarrow} Q' \stackrel{x}{\longrightarrow}$
implies $\exists Q' : Q \stackrel{\sigma}{\Longrightarrow} Q'$ and $\forall y \in L_U, y \neq x : Q' \stackrel{y}{\Longrightarrow\!\!\!\!\!\!/}$
implies Q **after** σ **refuses** $L_U \backslash \{x\}$

□

Lemma B.3
For $\sigma \in L^*$, $A \subseteq L_U$: Q_S **after** σ **must** A iff $\mathcal{O}_S(\sigma) \subseteq A$

□

Proof (lemma B.3)

We prove: Q_S **after** σ **refuses** A iff $\mathcal{O}_S(\sigma) \not\subseteq A$:

if: $\mathcal{O}_S(\sigma) \not\subseteq A$ implies $\delta \in \mathcal{O}_S(\sigma)$ or $\exists x \in L_U : x \in \mathcal{O}_S(\sigma)$ and $x \notin A$:

$\delta \in \mathcal{O}_S(\sigma)$:

$$\begin{array}{ll} & \delta \in \mathcal{O}_S(\sigma) \\ \text{implies} & Q_S \textbf{ after } \sigma \textbf{ refuses } L_U \\ \text{implies} & (* \text{ proposition 3.5.2 } *) \\ & Q_S \textbf{ after } \sigma \textbf{ refuses } A \end{array}$$

$\exists x \in L_U : x \in \mathcal{O}_S(\sigma)$ and $x \notin A$:

$$\begin{array}{ll} & \exists x \in L_U : x \in \mathcal{O}_S(\sigma) \text{ and } x \notin A \\ \text{implies} & \exists x : Q_S \stackrel{\sigma \cdot x}{\Longrightarrow} \text{ and } x \notin A \\ \text{implies} & (* \text{ proposition 5.12 } *) \\ & \exists x : \; Q_S \textbf{ after } \sigma \textbf{ refuses } L_U \backslash \{x\} \text{ and } x \notin A \\ \text{implies} & (* \text{ proposition 3.5.2 } *) \\ & Q_S \textbf{ after } \sigma \textbf{ refuses } A \end{array}$$

only if: Distinguish between Q_S **after** σ **refuses** L_U and $\neg\, Q_S$ **after** σ **refuses** L_U :

Q_S **after** σ **refuses** L_U :

$$\begin{array}{ll} & Q_S \textbf{ after } \sigma \textbf{ refuses } L_U \\ \text{implies} & \delta \in \mathcal{O}_S(\sigma) \\ \text{implies} & \mathcal{O}_S(\sigma) \not\subseteq A \end{array}$$

$\neg\, Q_S$ **after** σ **refuses** L_U :

$$\begin{array}{ll} & Q_S \textbf{ after } \sigma \textbf{ refuses } A \text{ and } \neg\, Q_S \textbf{ after } \sigma \textbf{ refuses } L_U \\ \text{implies} & (* \text{ proposition 3.5.1 } *) \\ & \exists Q' (Q_S \stackrel{\sigma}{\Longrightarrow} Q' \text{ and } \forall x \in A : Q' \stackrel{x}{\not\Longrightarrow}) \\ & \text{and } \forall Q'' (Q_S \stackrel{\sigma}{\Longrightarrow} Q'' \text{ implies } \exists y \in L_U : Q'' \stackrel{y}{\Longrightarrow}) \\ \text{implies} & \exists Q', \exists y \in L_U (Q_S \stackrel{\sigma}{\Longrightarrow} Q' \stackrel{y}{\Longrightarrow} \text{ and } \forall x \in A : Q' \stackrel{x}{\not\Longrightarrow}) \\ \text{implies} & \exists y \in \mathcal{O}_S(\sigma) : y \notin A \\ \text{implies} & \mathcal{O}_S(\sigma) \not\subseteq A \end{array}$$

□

Lemma B.4

$$\begin{array}{ll} & (\, \forall \sigma \in L^*, \forall A \subseteq L_U : \mathcal{O}_{S_1}(\sigma) \subseteq A \text{ implies } \mathcal{O}_{S_2}(\sigma) \subseteq A \,) \\ \text{iff} & (\, \forall \sigma \in L^* : \mathcal{O}_{S_2}(\sigma) \subseteq \mathcal{O}_{S_1}(\sigma) \,) \end{array}$$

□

Proof (lemma B.4)

if: Let $\sigma \in L^*$, $A \subseteq L_U$, with $\mathcal{O}_{S_1}(\sigma) \subseteq A$:
then $\mathcal{O}_{S_2}(\sigma) \subseteq \mathcal{O}_{S_1}(\sigma)$, thus $\mathcal{O}_{S_2}(\sigma) \subseteq A$.

only if: Let $\sigma \in L^*$, $x \in \mathcal{O}_{S_2}(\sigma)$, then we have to prove that $x \in \mathcal{O}_{S_1}(\sigma)$:

$x = \delta$: Take σ and $A = L_U$ in the left-hand side of the lemma:

$$\begin{array}{ll} & \mathcal{O}_{S_1}(\sigma) \subseteq L_U \text{ implies } \mathcal{O}_{S_2}(\sigma) \subseteq L_U \\ \text{iff} & \delta \notin \mathcal{O}_{S_1}(\sigma) \text{ implies } \delta \notin \mathcal{O}_{S_2}(\sigma) \\ \text{iff} & \delta \in \mathcal{O}_{S_2}(\sigma) \text{ implies } \delta \in \mathcal{O}_{S_1}(\sigma) \end{array}$$

thus $x \in \mathcal{O}_{S_1}(\sigma)$.

$x \in L_U, \delta \notin \mathcal{O}_{S_1}(\sigma)$: Take σ and $A = \mathcal{O}_{S_1}(\sigma)$ in the left-hand side of the lemma:

$$\begin{array}{rl} & \mathcal{O}_{S_1}(\sigma) \subseteq \mathcal{O}_{S_1}(\sigma) \text{ implies } \mathcal{O}_{S_2}(\sigma) \subseteq \mathcal{O}_{S_1}(\sigma) \\ \text{iff} & \mathcal{O}_{S_2}(\sigma) \subseteq \mathcal{O}_{S_1}(\sigma) \end{array}$$

thus, if $x \in \mathcal{O}_{S_2}(\sigma)$, then $x \in \mathcal{O}_{S_1}(\sigma)$.

$x \in L_U, \delta \in \mathcal{O}_{S_1}(\sigma)$: The left-hand side of the lemma is trivially fulfilled for σ and for all A, since $\mathcal{O}_{S_1}(\sigma) \not\subseteq A$. Now consider the trace $\sigma{\cdot}x$ and make the same distinction between $\delta \in \mathcal{O}_{S_1}(\sigma{\cdot}x)$ and $\delta \notin \mathcal{O}_{S_1}(\sigma{\cdot}x)$:

$\delta \in \mathcal{O}_{S_1}(\sigma{\cdot}x)$: This implies $\sigma{\cdot}x \in \mathit{traces}(Q_{S_1})$, thus $x \in \mathcal{O}_{S_1}(\sigma)$.

$\delta \notin \mathcal{O}_{S_1}(\sigma{\cdot}x)$: Take $\sigma{\cdot}x$ and $A = L_U$ in the left-hand side of the lemma, which implies $\delta \notin \mathcal{O}_{S_2}(\sigma{\cdot}x)$.
Since $x \in \mathcal{O}_{S_2}(\sigma)$: $\sigma{\cdot}x \in \mathit{traces}(Q_{S_2})$ and $\mathcal{O}_{S_2}(\sigma{\cdot}x) \neq \emptyset$.
Since $\delta \notin \mathcal{O}_{S_2}(\sigma{\cdot}x)$: $\exists y \in L_U$ with $y \in \mathcal{O}_{S_2}(\sigma{\cdot}x)$.
So we have $y \in \mathcal{O}_{S_2}(\sigma{\cdot}x)$, $y \in L_U$, and $\delta \notin \mathcal{O}_{S_1}(\sigma{\cdot}x)$, which allow to take $\sigma{\cdot}x$ and $A = \mathcal{O}_{S_1}(\sigma{\cdot}x)$ in the left-hand side of the lemma, analogous to the second item, implying $\mathcal{O}_{S_2}(\sigma{\cdot}x) \subseteq \mathcal{O}_{S_1}(\sigma{\cdot}x)$, thus $y \in \mathcal{O}_{S_1}(\sigma{\cdot}x)$.
Now we have $\sigma{\cdot}x{\cdot}y \in \mathit{traces}(Q_{S_1})$, thus $\sigma{\cdot}x \in \mathit{traces}(Q_{S_1})$, thus $x \in \mathcal{O}_{S_1}(\sigma)$.
□

Theorem 5.14

$$S_1 \approx_{\mathcal{Q}} S_2 \quad \text{iff} \quad \forall \sigma \in L^* : \mathcal{O}_{S_1}(\sigma) = \mathcal{O}_{S_2}(\sigma)$$

□

Proof (theorem 5.14)

$$\begin{array}{ll} & S_1 \approx_{\mathcal{Q}} S_2 \\ \text{iff} & (* \text{ equation (5.1) } *) \\ & \forall \sigma \in L^*, \forall A \subseteq L : \; Q_{S_1} \textbf{ after } \sigma \textbf{ must } A \;\; \text{iff} \;\; Q_{S_2} \textbf{ after } \sigma \textbf{ must } A \\ \text{iff} & (* \text{ proposition 5.11.2 } *) \\ & \forall \sigma \in L^*, \forall A \subseteq L_U : \; Q_{S_1} \textbf{ after } \sigma \textbf{ must } A \;\; \text{iff} \;\; Q_{S_2} \textbf{ after } \sigma \textbf{ must } A \\ \text{iff} & (* \text{ lemma B.3 } *) \\ & \forall \sigma \in L^*, \forall A \subseteq L_U : \mathcal{O}_{S_1}(\sigma) \subseteq A \;\text{ iff }\; \mathcal{O}_{S_2}(\sigma) \subseteq A \\ \text{iff} & (* \text{ lemma B.4 } *) \\ & \forall \sigma \in L^* : \mathcal{O}_{S_1}(\sigma) = \mathcal{O}_{S_2}(\sigma) \end{array}$$

□

Corollary 5.15

$$S_1 \approx_{\mathcal{Q}} S_2 \quad \text{iff} \quad \mathit{traces}(Q_{S_1}) = \mathit{traces}(Q_{S_2}) \;\text{ and }\; \delta\text{-}\mathit{traces}(S_1) = \delta\text{-}\mathit{traces}(S_2)$$

□

Proof (theorem 5.15)
First we prove:

$$(\forall \sigma \in L^*, \forall x \in L_U : Q_{S_1} \stackrel{\sigma\cdot x}{\Longrightarrow} \text{ iff } Q_{S_2} \stackrel{\sigma\cdot x}{\Longrightarrow}) \quad \text{iff} \quad (\forall \sigma \in L^* : Q_{S_1} \stackrel{\sigma}{\Longrightarrow} \text{ iff } Q_{S_2} \stackrel{\sigma}{\Longrightarrow})$$

if: Trivial.

only if: Let $Q_{S_1} \stackrel{\sigma}{\Longrightarrow}$, then we have to prove that $Q_{S_2} \stackrel{\sigma}{\Longrightarrow}$.
Distinguish between $\sigma \in L_I^*$, and $\sigma = \sigma' \cdot x \cdot \rho$ with $\sigma' \in L^*$, $x \in L_U$, and $\rho \in L_I^*$:

$\sigma \in L_I^*$: $Q_{S_2} \stackrel{\sigma}{\Longrightarrow}$ always holds.

$\sigma = \sigma' \cdot x \cdot \rho$: $Q_{S_1} \stackrel{\sigma' \cdot x \cdot \rho}{\Longrightarrow}$
implies $Q_{S_1} \stackrel{\sigma' \cdot x}{\Longrightarrow}$
implies $Q_{S_2} \stackrel{\sigma' \cdot x}{\Longrightarrow}$ ($*$ proposition 5.11.1 $*$)
implies $Q_{S_2} \stackrel{\sigma' \cdot x \cdot \rho}{\Longrightarrow}$

Now the proof of the theorem is straightforward:

$S_1 \approx_{\mathcal{Q}} S_2$
iff $\forall \sigma \in L^* : \mathcal{O}_{S_1}(\sigma) = \mathcal{O}_{S_2}(\sigma)$
iff $(\forall \sigma \in L^*, \forall x \in L_U : Q_{S_1} \stackrel{\sigma \cdot x}{\Longrightarrow}$ iff $Q_{S_2} \stackrel{\sigma \cdot x}{\Longrightarrow})$ and $(\forall \sigma \in L^* : \delta_{S_1}(\sigma) = \delta_{S_2}(\sigma))$
iff $(\forall \sigma \in L^* : Q_{S_1} \stackrel{\sigma}{\Longrightarrow}$ iff $Q_{S_2} \stackrel{\sigma}{\Longrightarrow})$ and $\delta\text{-}traces(S_1) = \delta\text{-}traces(S_2)$
iff $traces(Q_{S_1}) = traces(Q_{S_2})$ and $\delta\text{-}traces(S_1) = \delta\text{-}traces(S_2)$

□

B.5.2 Section 5.5 (Traces of Queue Contexts)

Proposition 5.18
Let $\sigma, \sigma_1, \sigma_2 \in L^*$:

1. $\sigma_1 \mathbin{@} \sigma_2$ implies $\sigma_1 \lceil L_U = \sigma_2 \lceil L_U$
2. $\sigma_1 \mathbin{@} \sigma_2$ implies $\sigma_1 \lceil L_I \preceq \sigma_2 \lceil L_I$
3. $\sigma \lceil L_U \mathbin{@} \sigma$
4. $\sigma_2 \backslash\backslash \sigma_1 = (\sigma_2 \lceil L_I) \backslash (\sigma_1 \lceil L_I)$

□

Proof (proposition 5.18)

1. By induction on the number of output actions in σ_1:
   - Basis: $\sigma_1 \in L_I^*$. By definition of @, also $\sigma_2 \in L_I^*$, and $\sigma_1 \lceil L_U = \epsilon = \sigma_2 \lceil L_U$.
   - Induction step: $\sigma_1 = \rho_1 \cdot x_1 \cdot \sigma_1'$, with $\rho_1 \in L_I^*, x_1 \in L_U, \sigma_1' \in L^*$.
     By definition of @, $\sigma_2 = \rho_2 \cdot x_2 \cdot \sigma_2'$, with $\rho_1 \preceq \rho_2, x_1 = x_2, \sigma_1' \mathbin{@} (\rho_2 \backslash \rho_1) \cdot \sigma_2'$.
     $\sigma_1 \lceil L_U = x_1 \cdot (\sigma_1' \lceil L_U) = (*$ induction $*) = x_1 \cdot ((\rho_2 \backslash \rho_1) \cdot \sigma_2' \lceil L_U) = (\rho_2 \cdot x_2 \cdot \sigma_2') \lceil L_U = \sigma_2 \lceil L_U$.
2. By induction on the number of output actions in σ_1 and σ_2; proposition 5.18.1 shows that the number of output actions in σ_1 and σ_2 is equal:
   - Basis: $\sigma_1, \sigma_2 \in L_I^*$. By definition of @: $\sigma_1 \preceq \sigma_2$, so $\sigma_1 \lceil L_I = \sigma_1 \preceq \sigma_2 = \sigma_2 \lceil L_I$.
   - Induction step:
     $\sigma_1 = \rho_1 \cdot x_1 \cdot \sigma_1', \sigma_2 = \rho_2 \cdot x_2 \cdot \sigma_2'$, with $\rho_1 \preceq \rho_2, x_1 = x_2, \sigma_1' \mathbin{@} (\rho_2 \backslash \rho_1) \cdot \sigma_2'$.

$\sigma_1 \lceil L_I = \rho_1 \cdot (\sigma_1' \lceil L_I) \preceq (*\ \text{induction}\ *) \preceq \rho_1 \cdot ((\rho_2 \backslash \rho_1) \cdot \sigma_2' \lceil L_I) = (\rho_2 \cdot \sigma_2') \lceil L_I = \sigma_2 \lceil L_I$.

3. By induction on the number of output actions in σ:
   - Basis: $\sigma \in L_I^*$. $\sigma \lceil L_U = \epsilon \preceq \sigma$, thus by definition of @: $\sigma \lceil L_U \mathbin{@} \sigma$.
   - Induction step: $\sigma = \rho \cdot x \cdot \sigma'$, with $\rho \in L_I^*, x \in L_U, \sigma' \in L^*$.
     By induction $(\rho \cdot \sigma') \lceil L_U @ (\rho \cdot \sigma')$, from which $\sigma' \lceil L_U @ \rho \cdot \sigma'$.
     Together with $\epsilon \preceq \rho$ and $x \lceil L_U = x$: $\sigma \lceil L_U = \epsilon \cdot x \cdot (\sigma' \lceil L_U) @ \rho \cdot x \cdot \sigma' = \sigma$.
4. By induction on the number of output actions in σ_1 and σ_2:
   - Basis: $\sigma_1, \sigma_2 \in L_I^*$: $\sigma_2 \backslash\backslash \sigma_1 = \sigma_2 \backslash \sigma_1 = (\sigma_2 \lceil L_I) \backslash (\sigma_1 \lceil L_I)$.
   - Induction step:
     $\sigma_1 = \rho_1 \cdot x_1 \cdot \sigma_1', \sigma_2 = \rho_2 \cdot x_2 \cdot \sigma_2'$, with $\rho_1 \preceq \rho_2, x_1 = x_2, \sigma_1' \mathbin{@} (\rho_2 \backslash \rho_1) \cdot \sigma_2'$.
     $\sigma_2 \backslash\backslash \sigma_1 = ((\rho_2 \backslash \rho_1) \cdot \sigma_2') \backslash\backslash \sigma_1' = (*\ \text{induction}\ *) = ((\rho_2 \backslash \rho_1) \cdot \sigma_2') \lceil L_I \backslash \sigma_1' \lceil L_I = \rho_1 \cdot (\rho_2 \backslash \rho_1) \cdot (\sigma_2') \lceil L_I \backslash \rho_1 \cdot (\sigma_1') \lceil L_I = \rho_2 \cdot (\sigma_2') \lceil L_I \backslash \rho_1 \cdot (\sigma_1') \lceil L_I = (\rho_2 \cdot x_2 \cdot \sigma_2') \lceil L_I \backslash (\rho_1 \cdot x_1 \cdot \sigma_1') \lceil L_I = (\sigma_2 \lceil L_I) \backslash (\sigma_1 \lceil L_I)$.

□

Proposition 5.20
$\langle L^*, @ \rangle$ is a well-founded poset.

□

Proof (proposition 5.20)
First we prove that @ is a partial order on L^*. The proof is given by induction on the number of output actions in σ_1. Note that if $\sigma_1 @ \sigma_2$, then σ_1 and σ_2 have the same number of output actions (proposition 5.18.1).

- Basis: suppose $\sigma_1, \sigma_2 \in L_I^*$, i.e. without output actions in σ_1, σ_2.
  Reflexivity, transitivity and anti-symmetry follow from reflexivity, transitivity and anti-symmetry of $\preceq$.
- Induction step: suppose $\sigma_1 = \rho_1 \cdot x_1 \cdot \sigma_1'$, $\sigma_2 = \rho_2 \cdot x_2 \cdot \sigma_2'$, $\sigma_3 = \rho_3 \cdot x_3 \cdot \sigma_3'$, with $\rho_1, \rho_2, \rho_3 \in L_I^*$, $x_1, x_2, x_3 \in L_U$, and $\sigma_1', \sigma_2', \sigma_3' \in L^*$. This implies that $\sigma_1', \sigma_2', \sigma_3'$ have one output action less than $\sigma_1, \sigma_2, \sigma_3$, thus according to the induction hypothesis we can assume the propositions to hold for $\sigma_1', \sigma_2', \sigma_3'$.

  Reflexivity of @ follows from the reflexivity of $\preceq$ and $=$, and from the induction hypothesis.

  Transitivity of @: suppose $\sigma_1 @ \sigma_2$ and $\sigma_2 @ \sigma_3$, then
  - $\rho_1 \preceq \rho_2$, $\rho_2 \preceq \rho_3$ imply $\rho_1 \preceq \rho_3$;
  - $x_1 = x_2$, $x_2 = x_3$ imply $x_1 = x_3$;

- from $\sigma_1 @ \sigma_2$: $\sigma_1' @ (\rho_2 \backslash \rho_1) \cdot \sigma_2'$ (1)
  from $\sigma_2 @ \sigma_3$: $\sigma_2' @ (\rho_3 \backslash \rho_2) \cdot \sigma_3'$ (2)
  using: if $\sigma_1 @ \sigma_2, \sigma_a \in L_I^*$ then $\sigma_a \cdot \sigma_1 @ \sigma_a \cdot \sigma_2$ (3)
  applying (3) to (2): $(\rho_2 \backslash \rho_1) \cdot \sigma_2' @ (\rho_2 \backslash \rho_1) \cdot (\rho_3 \backslash \rho_2) \cdot \sigma_3'$ (4)
  by induction from (1) and (4): $\sigma_1' @ (\rho_2 \backslash \rho_1) \cdot (\rho_3 \backslash \rho_2) \cdot \sigma_3'$ (5)
  and using the fact that $\rho_2 \backslash \rho_1 \cdot \rho_3 \backslash \rho_2 = \rho_3 \backslash \rho_1$ (6)
  we derive from (5) and (6): $\sigma_1' @ (\rho_3 \backslash \rho_1) \cdot \sigma_3'$

  Anti-symmetry follows from the anti-symmetry of $\preceq$ and $=$, and from the induction hypothesis, using the fact that now $\rho_2 \backslash \rho_1 = \rho_1 \backslash \rho_2 = \epsilon$.

Now we prove that @ is well-founded:

$$\forall T : \emptyset \neq T \subseteq L^* : \exists \sigma_m \in T : \forall \sigma \in T : \sigma @ \sigma_m \text{ implies } \sigma = \sigma_m$$

Assume non-well-foundedness:
$\exists T_0 : \emptyset \neq T_0 \subseteq L^* : \forall \sigma_m \in T_0 : \exists \rho_m \in T_0 : \rho_m @ \sigma_m$ and $\rho_m \neq \sigma_m$.
This means that, since $T_0 \neq \emptyset$, there is $\sigma_0 \in T_0$, and for this σ_0: $\exists \rho_1 \in T_0 : \rho_1 \overset{@}{\neq} \sigma_m$.
Again for ρ_1: $\exists \rho_2 \in T_0 : \rho_2 \overset{@}{\neq} \rho_1$, and so on. Hence there exists an infinite sequence of different traces

$$\ldots \overset{@}{\neq} \rho_4 \overset{@}{\neq} \rho_3 \overset{@}{\neq} \rho_2 \overset{@}{\neq} \rho_1 \overset{@}{\neq} \sigma_0 \tag{7}$$

Define two function $\alpha, \beta : L^* \to \mathbf{N}$, such that $\alpha(\sigma)$ gives the number of input actions in σ, and $\beta(\sigma)$ gives the sum of the position indices of the output actions. From the definition of @ it follows that $\rho_{i+1} \overset{@}{\neq} \rho_i$ implies that $\alpha(\rho_{i+1}) < \alpha(\rho_i)$ or $\beta(\rho_{i+1}) < \beta(\rho_i)$. This means that there exists a sequence

$$\ldots \cdot \langle \alpha(\rho_4), \beta(\rho_4) \rangle \cdot \langle \alpha(\rho_3), \beta(\rho_3) \rangle \cdot \langle \alpha(\rho_2), \beta(\rho_2) \rangle \cdot \langle \alpha(\rho_1), \beta(\rho_1) \rangle \cdot \langle \alpha(\sigma_0), \beta(\sigma_0) \rangle$$

with decreasing α or β. Since the length of σ_0 is finite, both the number of input actions in σ_0 and the sum of the numbers of output actions that follow each input action in σ_0, are finite: $\alpha(\sigma_0)$ and $\beta(\sigma_0)$ are finite, so this would imply the existence of an infinite sequence of pairs of decreasing natural numbers. Such a sequence does not exist (well-foundedness of $\mathbf{N}$), hence the sequence of traces (7) does not exist, hence the assumption of non-well-foundedness is not correct.

□

Proposition 5.22.1
If $\sigma_1 @ \sigma_2$ and $S \overset{\sigma_1}{\Longrightarrow} S'$ then $[_\epsilon \ll S \ll_\epsilon] \overset{\sigma_2}{\Longrightarrow} [_\epsilon \ll S' \ll_{\sigma_2 \backslash\backslash \sigma_1}]$.

□

Proof (proposition 5.22.1)
The proof is given by induction on the number of output actions in σ_1.

- Basis: suppose $\sigma_1, \sigma_2 \in L_I^*$.
  It follows that $\sigma_1 @ \sigma_2$ iff $\sigma_1 \preceq \sigma_2$, $\sigma_2 = \sigma_1 \cdot (\sigma_2 \backslash\backslash \sigma_1)$.
  Now we have:

$$\begin{array}{lll} & [_\epsilon \ll S \ll_\epsilon] & (*A1_Q*) \\ \overset{\sigma_1}{\Longrightarrow} & [_\epsilon \ll S \ll_{\sigma_1}] & (*I2_Q*) \\ \overset{\epsilon}{\Longrightarrow} & [_\epsilon \ll S' \ll_\epsilon] & (*A1_Q*) \\ \overset{\sigma_2 \backslash\backslash \sigma_1}{\Longrightarrow} & [_\epsilon \ll S' \ll_{\sigma_2 \backslash\backslash \sigma_1}] & \end{array}$$

- Induction step: suppose $\sigma_1 = \rho_1{\cdot}x_1{\cdot}\sigma_1'$, $\sigma_2 = \rho_2{\cdot}x_2{\cdot}\sigma_2'$, with $\rho_1, \rho_2 \in L_I^*$, $x_1, x_2 \in L_U$, and $\sigma_1', \sigma_2' \in L^*$.

  From the definition of $\sigma_1 @ \sigma_2$: $\rho_1 \preceq \rho_2,\ x_1 = x_2,\ \sigma_1' @ (\rho_2 \backslash \rho_1){\cdot}\sigma_2'$ (1)
  From the definition of $\sigma_2 \backslash\backslash \sigma_1$: $\sigma_2 \backslash\backslash \sigma_1 = ((\rho_2 \backslash \rho_1){\cdot}\sigma_2') \backslash\backslash \sigma_1'$ (2)
  Since $\rho_2 = \rho_1{\cdot}(\rho_2 \backslash \rho_1)$: $\sigma_2 = \rho_1{\cdot}(\rho_2 \backslash \rho_1){\cdot}x_1{\cdot}\sigma_2'$ (3)
  From $S \overset{\sigma_1}{\Longrightarrow} S'$: $\exists S_1, S_2 : S \overset{\rho_1}{\Longrightarrow} S_1 \overset{x_1}{\Longrightarrow} S_2 \overset{\sigma_1'}{\Longrightarrow} S'$ (4)

  Now we have: (5)
  $$\begin{array}{lll} & [_\epsilon{\ll}S{\ll}_\epsilon] & (*A1_Q, I2_Q, (4)*) \\ \overset{\rho_1}{\Longrightarrow} & [_\epsilon{\ll}S_1{\ll}_\epsilon] & (*I3_Q, (4)*) \\ \overset{\epsilon}{\Longrightarrow} & [_{x_1}{\ll}S_2{\ll}_\epsilon] & \end{array}$$

  Applying the induction hypothesis to σ_1' and $(\rho_2 \backslash \rho_1){\cdot}\sigma_2'$, using (1) and (4):

  $$\begin{array}{lll} & [_\epsilon{\ll}S_2{\ll}_\epsilon] \overset{(\rho_2\backslash\rho_1)\cdot\sigma_2'}{\Longrightarrow} & [_\epsilon{\ll}S'{\ll}_{((\rho_2\backslash\rho_1)\cdot\sigma_2')\backslash\backslash\sigma_1'}] \\ \text{implying} & [_{x_1}{\ll}S_2{\ll}_\epsilon] \overset{(\rho_2\backslash\rho_1)\cdot x_1\cdot\sigma_2'}{\Longrightarrow} & [_\epsilon{\ll}S'{\ll}_{((\rho_2\backslash\rho_1)\cdot\sigma_2')\backslash\backslash\sigma_1'}] \quad (6) \\ \text{Combining (5) and (6):} & [_\epsilon{\ll}S{\ll}_\epsilon] \overset{\rho_1}{\Longrightarrow} & [_{x_1}{\ll}S_2{\ll}_\epsilon] \\ & \overset{(\rho_2\backslash\rho_1)\cdot x_1\cdot\sigma_2'}{\Longrightarrow} & [_\epsilon{\ll}S'{\ll}_{((\rho_2\backslash\rho_1)\cdot\sigma_2')\backslash\backslash\sigma_1'}] \\ \text{thus, using (2) and (3):} & [_\epsilon{\ll}S{\ll}_\epsilon] \overset{\sigma_2}{\Longrightarrow} & [_\epsilon{\ll}S'{\ll}_{\sigma_2\backslash\backslash\sigma_1}] \end{array}$$

□

Proposition 5.22.2
If $[_\epsilon{\ll}S{\ll}_\epsilon] \overset{\sigma_2}{\Longrightarrow} [_\epsilon{\ll}S'{\ll}_{\sigma_r}]$ then $\exists \sigma_1 : S \overset{\sigma_1}{\Longrightarrow} S'$, $\sigma_1 @ \sigma_2$, and $\sigma_r = \sigma_2 \backslash\backslash \sigma_1$.

□

Proof (proposition 5.22.2)
The proof is given by induction on the number of output actions in σ_2.

- Basis: suppose $\sigma_2 \in L_I^*$.
  Since $[_\epsilon{\ll}S{\ll}_\epsilon] \overset{\sigma_2}{\Longrightarrow} [_\epsilon{\ll}S{\ll}_{\sigma_2}]$ always holds, $[_\epsilon{\ll}S{\ll}_\epsilon] \overset{\sigma_2}{\Longrightarrow} [_\epsilon{\ll}S'{\ll}_{\sigma_r}]$ immediately implies that there must be a prefix σ_1 of σ_2 ($\sigma_1 \preceq \sigma_2$), such that $S \overset{\sigma_1}{\Longrightarrow} S'$ and $\sigma_2 = \sigma_1{\cdot}\sigma_r$. σ_1 has the required properties.
- Induction step: suppose $\sigma_2 = \rho_2{\cdot}x_2{\cdot}\sigma_2'$, with $\rho_2 \in L_I^*$, $x_2 \in L_U$, and $\sigma_2' \in L^*$.
  We have: $[_\epsilon{\ll}S{\ll}_\epsilon] \overset{\rho_2\cdot x_2\cdot\sigma_2'}{\Longrightarrow} [_\epsilon{\ll}S'{\ll}_{\sigma_r}]$, implying that somewhere in this derivation the step $S_1 \xrightarrow{x_2} S_2$ occurs, for some S_1, S_2. (1)
  This implies that there are $\sigma_a, \rho_1 \in L_I^*$, with $\rho_1 \preceq \sigma_a \preceq \rho_2$ such that $S \overset{\rho_1}{\Longrightarrow} S_1$ and

  $$[_\epsilon{\ll}S{\ll}_\epsilon] \overset{\sigma_a}{\Longrightarrow} [_\epsilon{\ll}S_1{\ll}_{\sigma_a\backslash\rho_1}] \xrightarrow{\tau} [_{x_2}{\ll}S_2{\ll}_{\sigma_a\backslash\rho_1}] \overset{(\rho_2\backslash\sigma_a)\cdot x_2\cdot\sigma_2'}{\Longrightarrow} [_\epsilon{\ll}S'{\ll}_{\sigma_r}] \qquad (2)$$

  Taking the last part of this derivation:

$$
\begin{array}{rcl}
[_{x_2}\ll S_{2\ll\sigma_a\backslash\rho_1}] & \stackrel{(\rho_2\backslash\sigma_a)\cdot x_2\cdot\sigma_2'}{\Longrightarrow} & [_{\epsilon}\ll S'_{\ll\sigma_r}] \\
\text{implies } [_{\epsilon}\ll S_{2\ll\sigma_a\backslash\rho_1}] & \stackrel{(\rho_2\backslash\sigma_a)\cdot\sigma_2'}{\Longrightarrow} & [_{\epsilon}\ll S'_{\ll\sigma_r}] \\
\text{implies } [_{\epsilon}\ll S_{2\ll\epsilon}] & \stackrel{(\sigma_a\backslash\rho_1)\cdot(\rho_2\backslash\sigma_a)\cdot\sigma_2'}{\Longrightarrow} & [_{\epsilon}\ll S'_{\ll\sigma_r}] \\
\text{implies } [_{\epsilon}\ll S_{2\ll\epsilon}] & \stackrel{(\rho_2\backslash\rho_1)\cdot\sigma_2'}{\Longrightarrow} & [_{\epsilon}\ll S'_{\ll\sigma_r}]
\end{array}
$$

implies by induction

$$\exists\sigma_1' : S_2 \stackrel{\sigma_1'}{\Longrightarrow} S', \sigma_1' @ (\rho_2\backslash\rho_1)\cdot\sigma_2', \sigma_r = ((\rho_2\backslash\rho_1)\cdot\sigma_2')\backslash\backslash\sigma_1' = \sigma_2\backslash\backslash(\rho_1\cdot x_2\cdot\sigma_1') \quad (3)$$

Combining (1), (2), and (3):

$$S \stackrel{\rho_1}{\Longrightarrow} S_1 \xrightarrow{x_2} S_2 \stackrel{\sigma_1'}{\Longrightarrow} S'$$

with $\rho_1 \preceq \rho_2$, $\sigma_1' @ (\rho_2\backslash\rho_1)\cdot\sigma_2'$, hence $\rho_1\cdot x_2\cdot\sigma_1' @ \rho_2\cdot x_2\cdot\sigma_2'$.
Thus, the trace $\sigma_1 = \rho_1\cdot x_2\cdot\sigma_1'$ has the required properties.

□

Proposition 5.22.3

$$
\begin{array}{lll}
Q_S \stackrel{\sigma}{\Longrightarrow} & \text{iff} & \exists S', \sigma_i', \sigma_u' : [_{\epsilon}\ll S_{\ll\epsilon}] \stackrel{\sigma}{\Longrightarrow} [_{\sigma_u'}\ll S'_{\ll\sigma_i'}] \\
& \text{iff} & \exists S'', \sigma_i'' : [_{\epsilon}\ll S_{\ll\epsilon}] \stackrel{\sigma}{\Longrightarrow} [_{\epsilon}\ll S''_{\ll\sigma_i''}]
\end{array}
$$

□

Proof (proposition 5.22.3)

$Q_S \stackrel{\sigma}{\Longrightarrow}$ iff $\exists S', \sigma_i', \sigma_u' : [_{\epsilon}\ll S_{\ll\epsilon}] \stackrel{\sigma}{\Longrightarrow} [_{\sigma_u'}\ll S'_{\ll\sigma_i'}]$:
if follows from the definitions in table 1.1;
only if uses the fact that for any derivation $Q \stackrel{\sigma}{\Longrightarrow} Q'$, with $\sigma \in L^*$ and Q a queue context, Q' is a queue context too (proof of proposition 5.11.1).

$\exists S', \sigma_i', \sigma_u' : [_{\epsilon}\ll S_{\ll\epsilon}] \stackrel{\sigma}{\Longrightarrow} [_{\sigma_u'}\ll S'_{\ll\sigma_i'}]$ iff $\exists S'', \sigma_i'' : [_{\epsilon}\ll S_{\ll\epsilon}] \stackrel{\sigma}{\Longrightarrow} [_{\epsilon}\ll S''_{\ll\sigma_i''}]$:
only if: ∘ Let $\sigma \in L_I^*$: take $S'' = S$, and $\sigma_i'' = \sigma$.

∘ Let $\sigma = \sigma'\cdot x\cdot\rho$, with $\sigma' \in L^*$, $x \in L_U$, and $\rho \in L_I^*$, then somewhere in the left-hand side derivation the following steps must occur:

$$
\begin{array}{rcll}
[_{\sigma_v}\ll S_{1\ll\sigma_j}] & \xrightarrow{\tau} & [_{\sigma_v\cdot x}\ll S_{2\ll\sigma_j}] & (1) \\
[_{x\cdot\sigma_w}\ll S_{3\ll\sigma_k}] & \xrightarrow{x} & [_{\sigma_w}\ll S_{3\ll\sigma_k}] & (2)
\end{array}
$$

for some $\sigma_j, \sigma_k \in L_I^*$, $\sigma_v, \sigma_w \in L_U^*$, and S_1, S_2, S_3.

Let $\sigma_a \in L^*$ label the derivation from initial state to (1), and $\sigma_b \in L^*$ the derivation from (1) to (2), then we have:

$$
\begin{array}{rcl}
[_{\epsilon}\ll S_{\ll\epsilon}] & \stackrel{\sigma_a}{\Longrightarrow} & [_{\sigma_v}\ll S_{1\ll\sigma_j}] \\
& \xrightarrow{\tau} & [_{\sigma_v\cdot x}\ll S_{2\ll\sigma_j}] \\
& \stackrel{\sigma_b}{\Longrightarrow} & [_{x\cdot\sigma_w}\ll S_{3\ll\sigma_k}] \\
& \xrightarrow{x} & [_{\sigma_w}\ll S_{3\ll\sigma_k}] \\
& \stackrel{\rho}{\Longrightarrow} & [_{\sigma_u'}\ll S'_{\ll\sigma_i'}]
\end{array}
$$

It follows that $\sigma_v = \sigma_b \lceil L_U$ and that also the following derivation is possible:

$$[_{(\sigma_b \lceil L_U)\cdot x \ll} S_{2 \ll \sigma_j}] \overset{\sigma_b \cdot x \cdot \rho}{\Longrightarrow} [_{\epsilon \ll} S_{2 \ll \sigma_j \cdot (\sigma_b \lceil L_I) \cdot \rho}]$$

Consequently, $S'' = S_2$, and $\sigma_i'' = \sigma_j \cdot (\sigma_b \lceil L_I) \cdot \rho$ have the required properties.

if: Take $S' = S''$, $\sigma_i' = \sigma_i''$, and $\sigma_u' = \epsilon$.

□

Corollary 5.23

1. $\sigma_1 \mathbin{@} \sigma_2$ and $\sigma_1 \in \mathit{traces}(S)$ imply $\sigma_2 \in \mathit{traces}(Q_S)$
2. $\sigma_1 \mathbin{@} \sigma_2$ and $\sigma_1 \in \mathit{traces}(Q_S)$ imply $\sigma_2 \in \mathit{traces}(Q_S)$
3. $\sigma_2 \in \mathit{traces}(Q_S)$ implies $\exists \sigma_1 \in \mathit{traces}(S) : \sigma_1 \mathbin{@} \sigma_2$

□

Proof (corollary 5.23)

1. A direct consequence of proposition 5.22.1.
2. $\sigma_1 \in \mathit{traces}(Q_S)$ (∗ corollary 5.23.3 ∗)
   implies $\exists \sigma_0 \in \mathit{traces}(S) : \sigma_0 \mathbin{@} \sigma_1$ (∗ transitivity of @, $\sigma_1 \mathbin{@} \sigma_2$ ∗)
   implies $\exists \sigma_0 \in \mathit{traces}(S) : \sigma_0 \mathbin{@} \sigma_2$ (∗ corollary 5.23.1 ∗)
   implies $\sigma_2 \in \mathit{traces}(Q_S)$
3. $\sigma_2 \in \mathit{traces}(Q_S)$ (∗ proposition 5.22.3 ∗)
   implies $\exists S'', \sigma_i'' : Q_S \overset{\sigma_2}{\Longrightarrow} [_{\epsilon \ll} S''_{\ll \sigma_i''}]$ (∗ proposition 5.22.2 ∗)
   implies $\exists \sigma_1 \in \mathit{traces}(S) : \sigma_1 \mathbin{@} \sigma_2$

□

Proposition 5.25

1. $\mathit{tracks}(S) \subseteq \mathit{traces}(S) \subseteq \mathit{traces}(Q_S) \subseteq L^*$
2. $\mathit{traces}(Q_S) = \overline{\mathit{traces}(S)} = \overline{\mathit{tracks}(S)}$
3. $\mathit{tracks}(S) = \mathit{min}_{@}(\mathit{traces}(Q_S))$
4. A track is either equal to ϵ, or it ends with an output action.

□

Proof (proposition 5.25)

1. $\mathit{tracks}(S) \subseteq \mathit{traces}(S)$ by definition 5.24;
   $\mathit{traces}(S) \subseteq \mathit{traces}(Q_S)$ from corollary 5.23.1 and reflexivity of @;
   $\mathit{traces}(Q_S) \subseteq L^*$ by definition of *traces*.
2. $\mathit{traces}(Q_S) = \overline{\mathit{traces}(S)}$: from corollary 5.23.1 and 5.23.3;
   $\overline{\mathit{traces}(S)} = \overline{\mathit{tracks}(S)}$: using proposition A.2 with
   $\mathit{min}_{@}(\mathit{traces}(S)) = \mathit{tracks}(S) = \mathit{min}_{@}(\mathit{tracks}(S))$.
3. $\mathit{min}_{@}(\mathit{traces}(S)) = \mathit{min}_{@}(\mathit{traces}(Q_S))$: application of 5.25.2 (first part) to proposition A.1.

4. Assume a track σ ends with an input action: $\sigma = \sigma' \cdot a$. Then $\sigma \in traces(S)$, hence $\sigma' \in traces(S)$. Moreover $\sigma' @ \sigma$ and $\sigma' \neq \sigma$, hence σ is not minimal, which leads to the contradiction $\sigma \notin tracks(S)$.

□

Theorem 5.26

$$tracks(S_1) = tracks(S_2) \quad \text{iff} \quad traces(Q_{S_1}) = traces(Q_{S_2})$$

□

Proof (theorem 5.26)
From proposition A.2, using

- $\langle L^*, @ \rangle$ is a well-founded poset (∗ proposition 5.20 ∗);
- $traces(Q_S) = \overline{traces(S)}$ (∗ proposition 5.25.2 ∗);
- $tracks(S) = min_{@}(traces(S))$ (∗ definition 5.24 ∗)

□

B.5.3 Section 5.6 (Output Deadlocks of Queue Contexts)

Proposition 5.29

1. $\delta\text{-}temp(S)$ and $\delta\text{-}perm(S)$ form a partition of $\delta\text{-}traces(S)$
2. $$\begin{aligned} & \delta\text{-}traces(S_1) = \delta\text{-}traces(S_2) \\ \text{iff} \quad & \delta\text{-}temp(S_1) = \delta\text{-}temp(S_2) \ \text{ and } \ \delta\text{-}perm(S_1) = \delta\text{-}perm(S_2) \end{aligned}$$

□

Proof (proposition 5.29)
Follows immediately from the definitions of δ-*perm*, δ-*temp*, and δ-*traces* (definitions 5.28 and 5.13.2).

□

Proposition 5.31
$\delta\text{-}perm(S_1) = \delta\text{-}perm(S_2)$ iff $P\text{-}tracks(S_1) = P\text{-}tracks(S_2)$

□

Proof (proposition 5.31)
δ-*perm*(S) is right-closed with respect to @:
let $\sigma \in \delta\text{-}perm(S)$ and $\sigma \,@\, \sigma'$, then $\delta_S(\sigma')$ (∗ definition of δ-*perm* ∗);
let $\sigma'' \in L^*$, $\sigma' @ \sigma''$, then $\sigma @ \sigma''$ (∗ transitivity of @ ∗), so $\delta(\sigma'')$.
Together: $\delta_S(\sigma')$ and $\forall \sigma'' \in L^* : \sigma' @ \sigma''$ implies $\delta(\sigma'')$, imply $\sigma' \in \delta\text{-}perm(S)$.

The proposition follows by applying proposition A.2, using well-foundedness of $\langle L^*, @ \rangle$ (proposition 5.20), right-closedness of $\delta\text{-}perm(S_1)$ and $\delta\text{-}perm(S_2)$, and definition 5.30: $P\text{-}tracks(S) = min_{@}(\delta\text{-}perm(S))$.

□

Proposition 5.33

1. $\langle L^*, |@| \rangle$ is a well-founded poset.
2. $\sigma_1 \,@\, \sigma_2$ implies $\sigma_1 \cdot (\sigma_2 \backslash\backslash \sigma_1) \,|@|\, \sigma_2$

3. If $\sigma_1 \,|@|\, \sigma_2$ and $S \stackrel{\sigma_1}{\Longrightarrow} S'$ then $[_{\epsilon\ll} S_{\ll\epsilon}] \stackrel{\sigma_2}{\Longrightarrow} [_{\epsilon\ll} S'_{\ll\epsilon}]$
4. If $[_{\epsilon\ll} S_{\ll\epsilon}] \stackrel{\sigma_2}{\Longrightarrow} [_{\epsilon\ll} S'_{\ll\epsilon}]$ then $\exists \sigma_1 : S \stackrel{\sigma_1}{\Longrightarrow} S'$ and $\sigma_1 \,|@|\, \sigma_2$
5. If $\sigma_1 \,|@|\, \sigma_2$ and $Q_S \stackrel{\sigma_1}{\Longrightarrow} Q'$ then $Q_S \stackrel{\sigma_2}{\Longrightarrow} Q'$
6. If $\delta_S(\sigma_1)$ and $\sigma_1 \,|@|\, \sigma_2$ then $\delta_S(\sigma_2)$

□

Proof (proposition 5.33)

1. Follows from the fact that $|@|$ is the intersection of a well-founded partial order ($@$) and an equivalence relation ($|\cdot| = |\cdot|$).
2. The proof is given by induction on the number of output actions in σ_1:
   - Basis: suppose $\sigma_1 \in L_I^*$ and $\sigma_1 \,@\, \sigma_2$.
     Then $\sigma_1 \preceq \sigma_2$ and $\sigma_2 \backslash\backslash \sigma_1 = \sigma_2 \backslash \sigma_1$, so $\sigma_1 \cdot (\sigma_2 \backslash\backslash \sigma_1) = \sigma_1 \cdot (\sigma_2 \backslash \sigma_1) = \sigma_2$.
     Because of reflexivity of $|@|$ we conclude $\sigma_1 \cdot (\sigma_2 \backslash\backslash \sigma_1) \,|@|\, \sigma_2$.
   - Induction step: suppose $\sigma_1 = \rho_1 \cdot x_1 \cdot \sigma_1'$, $\sigma_2 = \rho_2 \cdot x_2 \cdot \sigma_2'$, (with $\rho_1, \rho_2 \in L_I^*$, $x_1, x_2 \in L_U, \sigma_1', \sigma_2' \in L^*$), and $\sigma_1 \,@\, \sigma_2$, then $\rho_1 \preceq \rho_2, x_1 = x_2, \sigma_1' \,@\, (\rho_2 \backslash \rho_1) \cdot \sigma_2', \sigma_2 \backslash\backslash \sigma_1 = ((\rho_2 \backslash \rho_1) \cdot \sigma_2') \backslash\backslash \sigma_1'$.
     Now: $\sigma_1 \cdot (\sigma_2 \backslash\backslash \sigma_1) = \rho_1 \cdot x_1 \cdot \sigma_1' \cdot (((\rho_2 \backslash \rho_1) \cdot \sigma_2') \backslash\backslash \sigma_1')$.
     Since σ_1' contains one output action less than σ_1, we can apply the induction hypothesis:

     $$\sigma_1' \,@\, (\rho_2 \backslash \rho_1) \cdot \sigma_2' \text{ implies } \sigma_1' \cdot (((\rho_2 \backslash \rho_1) \cdot \sigma_2') \backslash\backslash \sigma_1') \,|@|\, (\rho_2 \backslash \rho_1) \cdot \sigma_2'$$

     implying ($*$ definitions of $@$ and $|\cdot|$ $*$) :
     $\rho_1 \cdot x_1 \cdot \sigma_1' \cdot (((\rho_2 \backslash \rho_1) \cdot \sigma_2') \backslash\backslash \sigma_1') \,|@|\, \rho_1 \cdot x_1 \cdot (\rho_2 \backslash \rho_1) \cdot \sigma_2' \,|@|\, \rho_1 \cdot (\rho_2 \backslash \rho_1) \cdot x_1 \cdot \sigma_2' = \rho_2 \cdot x_2 \cdot \sigma_2' = \sigma_2$
     Because of transitivity of $|@|$ we conclude $\sigma_1 \cdot (\sigma_2 \backslash\backslash \sigma_1) \,|@|\, \sigma_2$.
3. Using proposition 5.22.1, together with the fact that $\sigma_1 \,|@|\, \sigma_2$ implies $\sigma_2 \backslash\backslash \sigma_1 = \epsilon$, which follows from proposition 5.33.2.
4. Using proposition 5.22.2, and $\sigma_1 \,@\, \sigma_2$ and $\sigma_2 \backslash\backslash \sigma_1 = \epsilon$ imply $|\sigma_1| = |\sigma_2|$, which follows from proposition 5.33.2.
5. We prove (1), which is more general. The proposition follows with $\sigma_i = \sigma_u = \rho = \epsilon$.
   Let $S \in \mathcal{LTS}$, Q' a queue-context, $\sigma_1, \sigma_2 \in L^*$, $\sigma_i, \rho \in L_I^*$, $\sigma_u \in L_U^*$, then

   $$[_{\sigma_u\ll} S_{\ll\sigma_i}] \stackrel{\sigma_1}{\Longrightarrow} Q' \text{ and } \sigma_1 \,|@|\, \rho \cdot \sigma_2 \text{ imply } [_{\sigma_u\ll} S_{\ll\sigma_i \cdot \rho}] \stackrel{\sigma_2}{\Longrightarrow} Q' \qquad (1)$$

   The proof is given by induction on the number of output actions in σ_1:
   - Basis: suppose $\sigma_1 \in L_I^*$ and $\sigma_1 \,|@|\, \rho \cdot \sigma_2$.
     Then $\sigma_1 = \rho \cdot \sigma_2$, hence
     $$\begin{array}{ll} & [_{\sigma_u\ll} S_{\ll\sigma_i}] \stackrel{\sigma_1}{\Longrightarrow} Q' \\ \text{iff} & [_{\sigma_u\ll} S_{\ll\sigma_i}] \stackrel{\rho\cdot\sigma_2}{\Longrightarrow} Q' \\ \text{implies} & [_{\sigma_u\ll} S_{\ll\sigma_i\cdot\rho}] \stackrel{\sigma_2}{\Longrightarrow} Q' \end{array}$$
   - Induction step: suppose $\sigma_1 = \rho_1 \cdot x_1 \cdot \sigma_1'$, $\sigma_2 = \rho_2 \cdot x_2 \cdot \sigma_2'$, (with $\rho_1, \rho_2 \in L_I^*$, $x_1, x_2 \in L_U, \sigma_1', \sigma_2' \in L^*$), and $\sigma_1 \,|@|\, \rho \cdot \sigma_2$, then ($*$definition $|@|$, $@$ $*$):

$\rho_1 \preceq \rho{\cdot}\rho_2,\ x_1 = x_2,\ \sigma_1' \mathbin{@} ((\rho{\cdot}\rho_2)\backslash\rho_1){\cdot}\sigma_2',$ and $|\sigma_1| = |\rho{\cdot}\sigma_2| = |\rho| + |\sigma_2|$.
Moreover:
$$\begin{aligned} & |\sigma_1'| \\ = \ & |\sigma_1| - |\rho_1| - |x_1| \\ = \ & (|\rho| + |\sigma_2|) - |\rho_1| - 1 \\ = \ & |\rho| + (|\rho_2| + |x_2| + |\sigma_2'|) - |\rho_1| - 1 \\ = \ & |((\rho{\cdot}\rho_2)\backslash\rho_1){\cdot}\sigma_2'| \end{aligned}$$
and therefore $\sigma_1' \,|@|\, ((\rho{\cdot}\rho_2)\backslash\rho_1){\cdot}\sigma_2'$.

$$[_{\sigma_u} \ll S_{\ll\sigma_i}] \stackrel{\sigma_1}{\Longrightarrow} Q'$$

$$\text{implies} \quad \exists S_1, \sigma_{i_1}, \sigma_{u_1} : \ [_{\sigma_u} \ll S_{\ll\sigma_i}] \ \stackrel{\rho_1}{\Longrightarrow} \ [_{x_1\cdot\sigma_{u_1}} \ll S_{1\ll\sigma_{i_1}}] \ \stackrel{x_1}{\longrightarrow} \ [_{\sigma_{u_1}} \ll S_{1\ll\sigma_{i_1}}] \ \stackrel{\sigma_1'}{\Longrightarrow} \ Q' \qquad (2)$$

$$\text{implies} \quad [_{\sigma_u} \ll S_{\ll\sigma_i}] \ \stackrel{\rho_1\cdot((\rho\cdot\rho_2)\backslash\rho_1)}{\Longrightarrow} \ [_{x_1\cdot\sigma_{u_1}} \ll S_{1\ll\sigma_{i_1}\cdot((\rho\cdot\rho_2)\backslash\rho_1)}] \ \stackrel{x_1}{\longrightarrow} \ [_{\sigma_{u_1}} \ll S_{1\ll\sigma_{i_1}\cdot((\rho\cdot\rho_2)\backslash\rho_1)}] \qquad (3)$$

Because $\sigma_1' \,|@|\, ((\rho{\cdot}\rho_2)\backslash\rho_1){\cdot}\sigma_2'$, and σ_1' contains one output action less then σ_1, the induction hypothesis can be applied to the last transition of (2):

$$[_{\sigma_{u_1}} \ll S_{1\ll\sigma_{i_1}\cdot((\rho\cdot\rho_2)\backslash\rho_1)}] \stackrel{\sigma_2'}{\Longrightarrow} Q' \qquad (4)$$

(3), (4), and $\rho_1{\cdot}((\rho{\cdot}\rho_2)\backslash\rho_1) = \rho{\cdot}\rho_2$ imply

$$[_{\sigma_u} \ll S_{\ll\sigma_i\cdot\rho}] \stackrel{\rho_2}{\Longrightarrow} [_{x_2\cdot\sigma_{u_1}} \ll S_{1\ll\sigma_{i_1}\cdot((\rho\cdot\rho_2)\backslash\rho_1)}] \stackrel{x_2\cdot\sigma_2'}{\Longrightarrow} Q'$$

6. $\delta_S(\sigma_1)$ iff $\exists Q' : Q_S \stackrel{\sigma_1}{\Longrightarrow} Q'$ and $\forall x \in L_U : Q' \stackrel{x}{\not\Longrightarrow}$.
   From proposition 5.33.5 it follows that $Q_S \stackrel{\sigma_2}{\Longrightarrow} Q'$,
   and thus $\exists Q'$ ($Q_S \stackrel{\sigma_2}{\Longrightarrow} Q'$ and $\forall x \in L_U : Q' \stackrel{x}{\not\Longrightarrow}$), so $\delta_S(\sigma_2)$.

□

Proposition 5.35
For $S_1, S_2 \in \mathcal{LTS}$ such that *P-tracks*(S_1) = *P-tracks*(S_2):
δ-*temp*$(S_1) = \delta$-*temp*(S_2) iff *T-tracks*(S_1) = *T-tracks*(S_2)

□

Proof (proposition 5.35)

only if: Follows immediately from the definition of *T-tracks*.

if: Let $\sigma \in \delta$-*temp*(S_1), then by well-foundedness of $|@|$: $\exists \sigma_0 \in$ *T-tracks*(S_1) : $\sigma_0 \,|@|\, \sigma$.
So $\sigma_0 \in$ *T-tracks*(S_2), so $\delta_{S_2}(\sigma_0)$, and since $\sigma_0 \,|@|\, \sigma$: $\delta_{S_2}(\sigma)$ ($*$ prop. 5.33.6 $*$).
This means that $\sigma \in \delta$-*traces*(S_2), so ($*$ proposition 5.29.1 $*$):
$\sigma \in \delta$-*temp*(S_2) or $\sigma \in \delta$-*perm*(S_2).
Since $\sigma \in \delta$-*temp*(S_1) and δ-*perm*$(S_1) = \delta$-*perm*(S_2) ($*$ proposition 5.31 $*$):
$\sigma \in \delta$-*temp*(S_2).

□

Theorem 5.36

$$S_1 \approx_{\mathcal{Q}} S_2 \quad \text{iff} \quad \begin{array}{ll} \mathit{tracks}(S_1) = \mathit{tracks}(S_2) & \text{and} \\ \mathit{P\text{-}tracks}(S_1) = \mathit{P\text{-}tracks}(S_2) & \text{and} \\ \mathit{T\text{-}tracks}(S_1) = \mathit{T\text{-}tracks}(S_2) & \end{array}$$

□

Proof (theorem 5.36)

$S_1 \approx_{\mathcal{Q}} S_2$
iff $(*$ theorem 5.15 $*)$
$\mathit{traces}(Q_{S_1}) = \mathit{traces}(Q_{S_2})$ and
$\delta\text{-}\mathit{traces}(S_1) = \delta\text{-}\mathit{traces}(S_2)$
iff $(*$ theorem 5.26 and proposition 5.29.2 $*)$
$\mathit{tracks}(S_1) = \mathit{tracks}(S_2)$ and
$\delta\text{-}\mathit{temp}(S_1) = \delta\text{-}\mathit{temp}(S_2)$ and
$\delta\text{-}\mathit{perm}(S_1) = \delta\text{-}\mathit{perm}(S_2)$
iff $(*$ propositions 5.31 and 5.35 $*)$
$\mathit{tracks}(S_1) = \mathit{tracks}(S_2)$ and
$\mathit{P\text{-}tracks}(S_1) = \mathit{P\text{-}tracks}(S_2)$ and
$\mathit{T\text{-}tracks}(S_1) = \mathit{T\text{-}tracks}(S_2)$

□

Proposition 5.38
Let $S \in \mathcal{LTS}$, $\sigma \in L^*$, then S **after** σ **refuses** L_U implies $\delta_S(\sigma)$.

□

Proof (proposition 5.38)

S **after** σ **refuses** L_U
implies $(*$ definition 3.4 $*)$
$\exists S' : S \stackrel{\sigma}{\Longrightarrow} S'$ and $\forall x \in L_U : S' \stackrel{x}{\Longrightarrow\!\!\!\!\!\!/\;\;}$
implies $(*$ $\sigma@\sigma$, proposition 5.22.1, definition 5.4 $*)$
$\exists S' : [_{\epsilon\ll} S_{\ll\epsilon}] \stackrel{\sigma}{\Longrightarrow} [_{\epsilon\ll} S'_{\ll\epsilon}]$ and $\forall x \in L_U : [_{\epsilon\ll} S'_{\ll\epsilon}] \stackrel{x}{\Longrightarrow\!\!\!\!\!\!/\;\;}$
implies $(*$ definition 5.4 $*)$
$\exists Q' : Q_S \stackrel{\sigma}{\Longrightarrow} Q'$ and $\forall x \in L_U : Q' \stackrel{x}{\Longrightarrow\!\!\!\!\!\!/\;\;}$
implies $(*$ definition 3.4 $*)$
Q_S **after** σ **refuses** L_U
implies $(*$ definition 5.13.2 $*)$
$\delta_S(\sigma)$

□

Proposition 5.39
Let $S \in \mathcal{LTS}$, $\sigma \in L^*$, $a \in L_I$, then

$$S \textbf{ after } \sigma \textbf{ refuses } \{a\} \cup L_U \quad \text{implies} \quad \sigma \cdot a \in \delta\text{-}\mathit{perm}(S)$$

□

Proof (proposition 5.39)

S **after** σ **refuses** $\{a\} \cup L_U$
implies $\exists S' : S \stackrel{\sigma}{\Longrightarrow} S'$ and $\forall b \in \{a\} \cup L_U : S' \stackrel{b}{\Longrightarrow\!\!\!\!\!\!/\;\;}$

implies $\exists S' : [_{\epsilon\ll} S_{\ll\epsilon}] \stackrel{\sigma\cdot a}{\Longrightarrow} [_{\epsilon\ll} S'_{\ll a}]$ and $\forall x \in L_U : [_{\epsilon\ll} S'_{\ll a}] \stackrel{x}{\Longrightarrow\!\!\!\!\!/}$ (1)

implies $\exists Q' : Q \stackrel{\sigma\cdot a}{\Longrightarrow} Q'$ and $\forall x \in L_U : Q' \stackrel{x}{\Longrightarrow\!\!\!\!\!/}$

implies $\delta_S(\sigma\cdot a)$

Moreover, it follows from (1) that for all $\rho \in L_I^*$:

$[_{\epsilon\ll} S'_{\ll a}] \stackrel{\rho}{\Longrightarrow} [_{\epsilon\ll} S'_{\ll a\cdot\rho}] \stackrel{x}{\Longrightarrow\!\!\!\!\!/}$ for any $x \in L_U$, so for all $\rho \in L_I^* : \delta_S(\sigma\cdot a\cdot\rho)$ (2)

Now we have to prove that $\sigma\cdot a \in \delta\text{-}perm$:
let $\sigma' \in L^*$, $\sigma\cdot a \,@\, \sigma'$, then $\sigma\cdot a\cdot(\sigma'\backslash\backslash(\sigma\cdot a)) \,|@|\, \sigma'$ ($*$ proposition 5.33.2 $*$),
and $\delta_S(\sigma\cdot a\cdot(\sigma'\backslash\backslash(\sigma\cdot a)))$ ($*$ $\sigma'\backslash\backslash(\sigma\cdot a) \in L_I^*$, and (2) $*$).
Using proposition 5.33.6 we conclude that $\delta_S(\sigma')$, so $\sigma\cdot a \in \delta\text{-}perm(S)$.

□

Proposition 5.41

1. $\delta\text{-}traces(S) \;=\; \delta\text{-}empty(S) \cup \delta\text{-}block(S)$
2. Not for all S: $\delta\text{-}empty(S) \cap \delta\text{-}block(S) \;=\; \emptyset$
3. $S_1 \approx_{\mathcal{Q}} S_2$ does not imply $\delta\text{-}empty(S_1) = \delta\text{-}empty(S_2)$,
   nor $\delta\text{-}block(S_1) = \delta\text{-}block(S_2)$

□

Proof (proposition 5.41)
To prove:

$\delta_S(\sigma)$ iff ($\exists \sigma' \in traces(S) : \sigma' \,|@|\, \sigma$ and S **after** σ' **refuses** L_U)
or ($\exists \sigma' \in traces(S), a \in L_I : \sigma'\cdot a \,@\, \sigma$ and S **after** σ' **refuses** $\{a\}\cup L_U$)

1. *if*: $\exists \sigma' \in traces(S) : \sigma' \,|@|\, \sigma$ and S **after** σ' **refuses** L_U :
   S **after** σ' **refuses** L_U implies $\delta_S(\sigma')$ ($*$ proposition 5.38 $*$);
   $\delta_S(\sigma')$ and $\sigma' \,|@|\, \sigma$ imply $\delta_S(\sigma)$ ($*$ proposition 5.33.6 $*$).

   $\exists \sigma' \in traces(S), a \in L_I : \sigma'\cdot a \,@\, \sigma$ and S **after** σ' **refuses** $\{a\}\cup L_U$:
   S **after** σ' **refuses** $\{a\}\cup L_U$ implies $\sigma'\cdot a \in \delta\text{-}perm(S)$ ($*$prop. 5.39$*$);
   $\sigma'\cdot a \in \delta\text{-}perm(S)$ and $\sigma'\cdot a \,@\, \sigma$ imply $\delta_S(\sigma)$ ($*$definition of $\delta\text{-}perm*$).

   only if:

   $\delta_S(\sigma)$
   implies $[_{\epsilon\ll} S_{\ll\epsilon}]$ **after** σ **refuses** L_U
   implies $\exists S', \sigma'_i, \sigma'_u : [_{\epsilon\ll} S_{\ll\epsilon}] \stackrel{\sigma}{\Longrightarrow} [_{\sigma'_u\ll} S'_{\ll\sigma'_i}]$ and $\forall x \in L_U : [_{\sigma'_u\ll} S'_{\ll\sigma'_i}] \stackrel{x}{\Longrightarrow\!\!\!\!\!/}$
   implies $\exists S', \sigma'_i : [_{\epsilon\ll} S_{\ll\epsilon}] \stackrel{\sigma}{\Longrightarrow} [_{\epsilon\ll} S'_{\ll\sigma'_i}]$ and $\forall x \in L_U : [_{\epsilon\ll} S'_{\ll\sigma'_i}] \stackrel{x}{\Longrightarrow\!\!\!\!\!/}$
   implies $\exists S', \sigma'_i : [_{\epsilon\ll} S_{\ll\epsilon}] \stackrel{\sigma}{\Longrightarrow} [_{\epsilon\ll} S'_{\ll\sigma'_i}]$ and $\forall \rho \preceq \sigma'_i, \forall x \in L_U : S' \stackrel{\rho\cdot x}{\Longrightarrow\!\!\!\!\!/}$

   Now distinguish between $S' \stackrel{\sigma'_i}{\Longrightarrow}$ and $S' \stackrel{\sigma'_i}{\Longrightarrow\!\!\!\!\!/}$:

   $S' \stackrel{\sigma'_i}{\Longrightarrow}$:

$$\begin{array}{rl}
& \exists S', \sigma'_i : [_\epsilon\!\ll S \ll_\epsilon] \stackrel{\sigma}{\Longrightarrow} [_\epsilon\!\ll S' \ll_{\sigma'_i}] \\
& \quad \text{and } \forall \rho \preceq \sigma'_i, \forall x \in L_U : S' \stackrel{\rho\cdot x}{\Longrightarrow\!\!\!\!\!\!/} \\
\text{implies} & \exists S', \sigma'_i : [_\epsilon\!\ll S \ll_\epsilon] \stackrel{\sigma}{\Longrightarrow} [_\epsilon\!\ll S' \ll_{\sigma'_i}] \\
& \quad \text{and } \exists S'' : S' \stackrel{\sigma'_i}{\Longrightarrow} S'' \text{ and } \forall x \in L_U : S'' \stackrel{x}{\Longrightarrow\!\!\!\!\!\!/} \\
\text{implies} & \exists S'' : [_\epsilon\!\ll S \ll_\epsilon] \stackrel{\sigma}{\Longrightarrow} [_\epsilon\!\ll S'' \ll_\epsilon] \text{ and } \forall x \in L_U : S'' \stackrel{x}{\Longrightarrow\!\!\!\!\!\!/} \\
\text{implies} & \exists S'', \sigma' : S \stackrel{\sigma'}{\Longrightarrow} S'' \text{ and } \sigma' \,|@|\, \sigma \text{ and } \forall x \in L_U : S'' \stackrel{x}{\Longrightarrow\!\!\!\!\!\!/} \\
\text{implies} & \exists \sigma' \in \mathit{traces}(S) : \sigma' \,|@|\, \sigma \text{ and } S \textbf{ after } \sigma' \textbf{ refuses } L_U
\end{array}$$

$S' \stackrel{\sigma'_i}{\Longrightarrow\!\!\!\!\!\!/}$:

$$\begin{array}{rl}
\text{then} & \exists a \in L_I, \rho_1, \rho_2 \in L_I^*, S'' : \sigma'_i = \rho_1 \cdot a \cdot \rho_2 \text{ and } S' \stackrel{\rho_1}{\Longrightarrow} S'' \stackrel{a}{\Longrightarrow\!\!\!\!\!\!/} \\
\text{Now:} & \exists S', \sigma'_i : [_\epsilon\!\ll S \ll_\epsilon] \stackrel{\sigma}{\Longrightarrow} [_\epsilon\!\ll S' \ll_{\sigma'_i}] \\
& \quad \text{and } \forall \rho \preceq \sigma'_i, \forall x \in L_U : S' \stackrel{\rho\cdot x}{\Longrightarrow\!\!\!\!\!\!/} \\
\text{implies} & \exists a \in L_I, \rho_1, \rho_2 \in L_I^*, S', S'' : \\
& [_\epsilon\!\ll S \ll_\epsilon] \stackrel{\sigma}{\Longrightarrow} [_\epsilon\!\ll S'' \ll_{a\cdot\rho_2}] \quad \text{and} \\
& \forall x \in L_U : S' \stackrel{\rho_1\cdot x}{\Longrightarrow\!\!\!\!\!\!/} \quad \text{and} \\
& S' \stackrel{\rho_1}{\Longrightarrow} S'' \stackrel{a}{\Longrightarrow\!\!\!\!\!\!/} \\
\text{implies} & \exists a \in L_I, \rho_1, \rho_2 \in L_I^*, S'', \sigma' : \\
& S \stackrel{\sigma'}{\Longrightarrow} S'' \quad \text{and} \\
& \sigma' \,@\, \sigma \quad \text{and} \\
& a \cdot \rho_2 = \sigma \backslash\backslash \sigma' \quad \text{and} \\
& \forall \mu \in \{a\} \cup L_U : S'' \stackrel{\mu}{\Longrightarrow\!\!\!\!\!\!/} \\
\text{implies} & \exists \sigma' \in \mathit{traces}(S), a \in L_I : \\
& \sigma' \cdot a \,@\, \sigma \text{ and } S \textbf{ after } \sigma' \textbf{ refuses } \{a\} \cup L_U
\end{array}$$

2. See example 5.43.
3. See example 5.43.

□

Proposition 5.44

1. $\delta\text{-}\mathit{block}(S) \subseteq \delta\text{-}\mathit{perm}(S)$
2. $\delta\text{-}\mathit{temp}(S) \subseteq \delta\text{-}\mathit{empty}(S)$
3. $T\text{-}\mathit{tracks}(S) = E\text{-}\mathit{tracks}(S) \cap \delta\text{-}\mathit{temp}(S)$ (and hence $T\text{-}\mathit{tracks}(S) \subseteq E\text{-}\mathit{tracks}(S)$)
4. $\sigma \in T\text{-}\mathit{tracks}(S)$ implies S **after** σ **refuses** L_U (and hence $\sigma \in \mathit{traces}(S)$)
5. $\sigma \in P\text{-}\mathit{tracks}(S)$ implies
   $\sigma \in \delta\text{-}\mathit{empty}(S)$
   or ($\exists \sigma' \in \mathit{traces}(S), a \in L_I : \sigma' \cdot a = \sigma$ and S **after** σ' **refuses** $\{a\} \cup L_U$)
6. $\sigma \in \delta\text{-}\mathit{perm}(S)$ iff $\delta_S(\sigma)$ and $\forall a \in L_I : \sigma \cdot a \in \delta\text{-}\mathit{perm}(S)$

□

Proof (proposition 5.44)

1. $\sigma \in \delta\text{-}block(S)$
   implies $\exists \sigma' \in traces(S), a \in L_I : \sigma' \cdot a \,@\, \sigma$ and S **after** σ' **refuses** $\{a\} \cup L_U$
   implies $(*$ proposition 5.39 $*)$
   $\sigma' \cdot a \in \delta\text{-}perm(S)$ and $\sigma' \cdot a \,@\, \sigma$
   implies $(*$ definition 5.28.2 $*)$
   $\sigma \in \delta\text{-}perm(S)$

2. $\sigma \in \delta\text{-}temp(S)$
   implies $(*$ proposition 5.29.1 $*)$
   $\sigma \notin \delta\text{-}perm(S)$
   implies $(*$ proposition 5.44.1 $*)$
   $\sigma \notin \delta\text{-}block(S)$
   implies $(*$ proposition 5.41.1 $*)$
   $\sigma \in \delta\text{-}empty(S)$

3. $\subseteq$: Let $\sigma \in T\text{-}tracks(S)$ then $\sigma \in \delta\text{-}temp(S)$ (definition 5.34). Moreover, we have to prove that $\sigma \in E\text{-}tracks(S)$, which means that (definition 5.40.3):

$$\begin{array}{l} S \textbf{ after } \sigma \textbf{ refuses } L_U \quad \text{and} \\ (\forall \sigma'' :\ S \textbf{ after } \sigma'' \textbf{ refuses } L_U \text{ and } \sigma'' \,|@|\, \sigma \text{ implies } \sigma'' = \sigma) \end{array} \tag{1}$$

$$\sigma \in T\text{-}tracks(S) \text{ implies } \sigma \in \delta\text{-}temp(S) \tag{2}$$
$$\text{and } \forall \sigma' \in \delta\text{-}temp(S) : \sigma' \,|@|\, \sigma \text{ implies } \sigma' = \sigma \tag{3}$$
$$\text{and } \exists \sigma_1 : \sigma \,@\, \sigma_1 \text{ and not } \delta_S(\sigma_1) \tag{4}$$

Using (2), (3), and (4) the two parts of (1) are proved:

S **after** σ **refuses** L_U :
(2)
implies $(*$ proposition 5.44.2 $*)$
$\sigma \in \delta\text{-}empty(S)$
implies $(*$ definition 5.40.1 $*)$
$\exists \sigma_2 \in traces(S) : \sigma_2 \,|@|\, \sigma$ and S **after** σ_2 **refuses** L_U
implies $(*$ proposition 5.38 $*)$
$\delta_S(\sigma_2)$
implies $(*\ \sigma_2 \,|@|\, \sigma \,@\, \sigma_1$, so $\sigma_2 \,@\, \sigma_1$, (4) $*)$
$\sigma_2 \in \delta\text{-}temp(S)$
implies $(*\ \sigma_2 \,|@|\, \sigma$, (3) $*)$
$\sigma_2 = \sigma$
implies $(*\ S$ **after** σ_2 **refuses** $L_U\ *)$
S **after** σ **refuses** L_U

$\forall \sigma'' :\ S$ **after** σ'' **refuses** L_U and $\sigma'' \,|@|\, \sigma$ implies $\sigma'' = \sigma$:
Let $\sigma'' \in L^*$, such that S **after** σ'' **refuses** L_U and $\sigma'' \,|@|\, \sigma$, then

S **after** σ'' **refuses** L_U
implies $(*$ proposition 5.38 $*)$
$\delta_S(\sigma'')$
implies $(*$ $\sigma'' \mathbin{@} \sigma$, (2) $*)$
$\sigma'' \in \delta\text{-}temp(S)$
implies $(*$ (3) $*)$
$\sigma'' = \sigma$

$\supseteq$: Let $\sigma \in E\text{-}tracks(S) \cap \delta\text{-}temp$, then we have to prove

$$\sigma \in \delta\text{-}temp(S) \text{ and } \forall \sigma'' \in \delta\text{-}temp(S) : \sigma'' \,|@|\, \sigma \text{ implies } \sigma'' = \sigma$$

$\sigma \in \delta\text{-}temp(S)$: Immediately.

$\forall \sigma'' \in \delta\text{-}temp(S) : \sigma'' \,|@|\, \sigma$ implies $\sigma'' = \sigma$:
Let $\sigma'' \in \delta\text{-}temp(S)$ and $\sigma'' \,|@|\, \sigma$, $(*$ prop. 5.44.2 $*)$
then $\sigma'' \in \delta\text{-}empty(S)$.
By definition 5.40.1:
$\exists \sigma_1 \in traces(S) : \sigma_1 \,|@|\, \sigma''$ and S **after** σ_1 **refuses** L_U,
hence $\sigma_1 \,|@|\, \sigma'' \,|@|\, \sigma$.
Since $\sigma \in E\text{-}tracks(S)$:
$\forall \sigma' \in L^*$: S **after** σ' **refuses** L_U and $\sigma' \,|@|\, \sigma$ implies $\sigma' = \sigma$;
we conclude $\sigma_1 = \sigma'' = \sigma$.

4. Directly from proposition 5.44.3 and definition 5.40.3.

5. $\sigma \in P\text{-}tracks(S)$ implies $\sigma \in \delta\text{-}empty(S)$ or $\sigma \in \delta\text{-}block(S)$ $(*$ prop. 5.41.1 $*)$.
If $\sigma \in \delta\text{-}block(S)$ then
$\exists \sigma' \in traces(S), a \in L_I : \sigma' \cdot a \mathbin{@} \sigma$ and S **after** σ' **refuses** $\{a\} \cup L_U$
implying $\sigma' \cdot a \in \delta\text{-}perm(S)$ $(*$ proposition 5.39 $*)$
By definition of $P\text{-}tracks$, $\forall \sigma'' \in \delta\text{-}perm(S) : \sigma'' \mathbin{@} \sigma$ implies $\sigma'' = \sigma$, so $\sigma' a = \sigma$.

6. *only if*: Straightforward from the fact that $\delta\text{-}perm(S)$ is right-closed with respect to @, using $\sigma \mathbin{@} \sigma \cdot a$.

if: Let $\delta_S(\sigma)$ and $\forall a \in L_I : \sigma \cdot a \in \delta\text{-}perm(S)$, then we have to prove that $\sigma \mathbin{@} \sigma'$ implies $\delta_S(\sigma')$ for arbitrary σ'. Let $\sigma' \in L^*$, such that $\sigma \mathbin{@} \sigma'$, and distinguish between $|\sigma| = |\sigma'|$ and $|\sigma| \neq |\sigma'|$:

$|\sigma| = |\sigma'|$: $\sigma \,|@|\, \sigma'$ implies $\delta_S(\sigma')$ $(*$ $\delta_S(\sigma)$, proposition 5.33.6 $*)$.

$|\sigma| \neq |\sigma'|$: now $\exists a \in L_I, \sigma_i \in L_I^* : \sigma' \backslash\backslash \sigma = a \cdot \sigma_i$.
We have $(*$ proposition 5.33.2 $*) : \sigma a \mathbin{@} \sigma a \sigma_i \,|@|\, \sigma'$ and $\sigma a \in \delta\text{-}perm(S)$,
which implies $\delta_S(\sigma')$.

□

B.5.4 Section 5.7 (Queue Implementation Relations)

Proposition 5.51.1 (Equalities)

1. $\leq_{te} \;=\; \leq_{tr} \cap$ **conf**

2. $\leq_{\mathcal{O}} \;=\; \leq_{te}^{Q}$
3. $\leq_{\mathcal{O}} \;=\; \leq_{tr}^{Q} \cap \mathbf{conf}_Q$
4. $\leq_{\mathcal{O}} \;=\; \leq_{outputs} \cap \leq_{\delta}$
5. $\leq_{tr}^{Q} \;=\; \leq_{outputs}$

□

Proof (proposition 5.51.1)

5.51.1.1: Proposition 3.13.1.

5.51.1.2: Analogous to the proof of theorem 5.14:

$I \leq_{te}^{Q} S$
iff $(*$ definition 5.49.1 $*)$
$Q_I \leq_{te} Q_S$
iff $(*$ theorem 3.9 $*)$
$\forall \sigma \in L^*, \forall A \subseteq L :\; Q_S \textbf{ after } \sigma \textbf{ must } A \;\text{ implies }\; Q_I \textbf{ after } \sigma \textbf{ must } A$
iff $(*$ proposition 5.11.2 $*)$
$\forall \sigma \in L^*, \forall A \subseteq L_U :\; Q_S \textbf{ after } \sigma \textbf{ must } A \;\text{ implies }\; Q_I \textbf{ after } \sigma \textbf{ must } A$
iff $(*$ lemma B.3 $*)$
$\forall \sigma \in L^*, \forall A \subseteq L_U : \mathcal{O}_S(\sigma) \subseteq A \;\text{ implies }\; \mathcal{O}_I(\sigma) \subseteq A$
iff $(*$ lemma B.4 $*)$
$\forall \sigma \in L^* : \mathcal{O}_I(\sigma) \subseteq \mathcal{O}_S(\sigma)$

5.51.1.3: Directly from definition 5.49 and propositions 5.51.1.2 and 5.51.1.1.

5.51.1.4: Directly from definitions 5.13.3 and 5.50.

5.51.1.5: To be proved:

$$traces(Q_I) \subseteq traces(Q_S) \;\text{ iff }\; \forall \sigma \in L^* : outputs_I(\sigma) \subseteq outputs_S(\sigma)$$

only-if: Let $\sigma \in L^*$, $x \in outputs_I(\sigma)$, then, according to definition 5.13.1:

$Q_I \stackrel{\sigma \cdot x}{\Longrightarrow}$
implies $Q_S \stackrel{\sigma \cdot x}{\Longrightarrow}$
implies $x \in outputs_S(\sigma)$

if: First, let $\sigma \in traces(Q_I)$ and $\sigma \in L_I^*$:
according to proposition 5.11.1, $\sigma \in traces(Q_S)$ always holds.
Secondly, let $\sigma \in traces(Q_I)$ and $\sigma = \sigma' \cdot x \cdot \rho$, with $\sigma' \in L^*$, $x \in L_U$, $\rho \in L_I^*$:

$Q_I \stackrel{\sigma' \cdot x \cdot \rho}{\Longrightarrow}$ $(*$ table 1.1 $*)$
implies $Q_I \stackrel{\sigma' \cdot x}{\Longrightarrow}$ $(*$ definition 5.13.1 $*)$
implies $x \in outputs_I(\sigma')$ $(*$ premiss $*)$
implies $x \in outputs_S(\sigma')$ $(*$ definition 5.13.1 $*)$
implies $Q_S \stackrel{\sigma' \cdot x}{\Longrightarrow}$ $(*$ proposition 5.11.1 $*)$
implies $Q_S \stackrel{\sigma' \cdot x \cdot \rho}{\Longrightarrow}$ $(*$ definition 1.9 $*)$
implies $\sigma \in traces(Q_S)$

□

Proposition 5.51.2 (Inclusions)

1. $\leq_{te} \subseteq \leq_{tr}$
2. $\leq_{te} \subseteq$ **conf**
3. $\leq_{te} \subseteq \leq_{\mathcal{O}}$
4. $\leq_{tr} \subseteq \leq_{tr}^{Q}$
5. $\leq_{\mathcal{O}} \subseteq \leq_{tr}^{Q}$
6. $\leq_{\mathcal{O}} \subseteq \leq_{\delta}$
7. $\leq_{\mathcal{O}} \subseteq$ **conf**$_Q$

□

Proof (proposition 5.51.2)

5.51.2.1: Directly from proposition 5.51.1.1.

5.51.2.2: Directly from proposition 5.51.1.1.

5.51.2.3: To be proved:

$$\begin{array}{ll} & \forall \sigma \in L^*, \forall A \subseteq L : \ I \textbf{ after } \sigma \textbf{ refuses } A \text{ implies } S \textbf{ after } \sigma \textbf{ refuses } A \\ \text{implies} & \forall \sigma \in L^* : \mathcal{O}_I(\sigma) \subseteq \mathcal{O}_S(\sigma) \end{array}$$

Let $\sigma \in L^*$, $x \in \mathcal{O}_I(\sigma)$, then $x = \delta$, or $x \in L_U$:

$x = \delta$: This means that $\delta_I(\sigma)$ holds; according to proposition 5.41.1 one of the following cases holds:

(1) $\exists \sigma' \in \mathit{traces}(I) : \sigma' \,|@|\, \sigma$ and $I \textbf{ after } \sigma' \textbf{ refuses } L_U$
implying ($*$ 5.51.2.1: $\mathit{traces}(I) \subseteq \mathit{traces}(S)$ and
$I \textbf{ after } \sigma' \textbf{ refuses } L_U$ implies $S \textbf{ after } \sigma' \textbf{ refuses } L_U$ $*$)
$\exists \sigma' \in \mathit{traces}(S) : \sigma' \,|@|\, \sigma$ and $S \textbf{ after } \sigma' \textbf{ refuses } L_U$
implies $\delta_S(\sigma)$
implies $\delta \in \mathcal{O}_S(\sigma)$

(2) $\exists \sigma' \in \mathit{traces}(I), a \in L_I : \sigma' \cdot a \,@\, \sigma$ and $I \textbf{ after } \sigma' \textbf{ refuses } \{a\} \cup L_U$
implying ($*$ 5.51.2.1: $\mathit{traces}(I) \subseteq \mathit{traces}(S)$ and
$I \textbf{ after } \sigma' \textbf{ refuses } \{a\} \cup L_U$ implies
$S \textbf{ after } \sigma' \textbf{ refuses } \{a\} \cup L_U$ $*$)
$\exists \sigma' \in \mathit{traces}(S), a \in L_I :$
$\sigma' \cdot a \,@\, \sigma$ and $S \textbf{ after } \sigma' \textbf{ refuses } \{a\} \cup L_U$
implies $\delta_S(\sigma)$
implies $\delta \in \mathcal{O}_S(\sigma)$

$x \in L_U$:
$x \in \mathcal{O}_I(\sigma)$ and $x \in L_U$ ($*$ definition 5.13 $*$)
implies $\sigma \cdot x \in \mathit{traces}(Q_I)$ ($*$ corollary 5.23.3 $*$)
implies $\exists \sigma' \in \mathit{traces}(I) : \sigma' @\, \sigma \cdot x$ ($*$ $\mathit{traces}(I) \subseteq \mathit{traces}(S)$ $*$)
implies $\exists \sigma' \in \mathit{traces}(S) : \sigma' @\, \sigma \cdot x$ ($*$ corollary 5.23.1 $*$)
implies $\sigma \cdot x \in \mathit{traces}(Q_S)$ ($*$ definition 5.13 $*$)
implies $x \in \mathcal{O}_S(\sigma)$

5.51.2.4: To be proved: $traces(I) \subseteq traces(S)$ implies $traces(Q_I) \subseteq traces(Q_S)$
Let $\sigma \in traces(Q_I)$, then ($*$ corollary 5.23.3 $*$):
$\exists \sigma' \in traces(I) : \sigma' @ \sigma$ ($*$ premiss $*$)
implies $\exists \sigma' \in traces(S) : \sigma' @ \sigma$ ($*$ corollary 5.23.1 $*$)
implies $\sigma \in traces(Q_S)$

5.51.2.5: Directly from proposition 5.51.1.3.

5.51.2.6: Directly from proposition 5.51.1.4.

5.51.2.7: Directly from proposition 5.51.1.3.

□

Proposition 5.51.3 (Inequalities)

1. $\leq_{te} \neq \leq_{tr}$
2. $\leq_{te} \neq \mathbf{conf}$
3. $\leq_{te} \neq \leq_{\mathcal{O}}$
4. $\leq_{tr} \neq \leq_{tr}^{Q}$
5. $\leq_{\mathcal{O}} \neq \leq_{tr}^{Q}$
6. $\leq_{\mathcal{O}} \neq \leq_{\delta}$
7. $\leq_{\mathcal{O}} \neq \mathbf{conf}_Q$

□

Proof (proposition 5.51.3)
Figure B.1:

5.51.3.1: $S_1 \not\leq_{te} S_2$ and $S_1 \leq_{tr} S_2$.

5.51.3.2: $S_2 \not\leq_{te} S_1$ and S_2 **conf** S_1.

5.51.3.3: $S_1 \not\leq_{te} S_2$ and $S_1 \leq_{\mathcal{O}} S_2$.

5.51.3.4: $S_6 \not\leq_{tr} S_5$ and $S_6 \leq_{tr}^{Q} S_5$.

5.51.3.5: $S_4 \not\leq_{\mathcal{O}} S_1$ and $S_4 \leq_{tr}^{Q} S_1$.

5.51.3.6: $S_8 \not\leq_{\mathcal{O}} S_7$ and $S_8 \leq_{\delta} S_7$.

5.51.3.7: $S_3 \not\leq_{\mathcal{O}} S_1$ and S_3 $\mathbf{conf}_Q S_1$.

□

Proposition 5.51.4 (Distinctions)

1. $\leq_{tr} \neq$ **conf**
2. $\leq_{tr} \neq \leq_{\mathcal{O}}$
3. **conf** $\neq \leq_{\mathcal{O}}$
4. **conf** $\neq \mathbf{conf}_Q$
5. **conf** $\neq \leq_{\delta}$
6. **conf** $\neq \leq_{tr}^{Q}$

7. $\leq_{tr}^{Q} \neq \mathbf{conf}_Q$
8. $\leq_{tr}^{Q} \neq \leq_\delta$
9. $\mathbf{conf}_Q \neq \leq_\delta$

□

Proof (proposition 5.51.4)
Figure B.1:

5.51.4.1: $S_2 \not\leq_{tr} S_1$ and S_2 **conf** S_1; $S_1 \leq_{tr} S_2$ and S_1 **co$\not$nf** S_2.
5.51.4.2: $S_6 \not\leq_{tr} S_7$ and $S_6 \leq_{\mathcal{O}} S_7$; $S_7 \leq_{tr} S_5$ and $S_7 \not\leq_{\mathcal{O}} S_5$.
5.51.4.3: S_1 **co$\not$nf** S_2 and $S_1 \leq_{\mathcal{O}} S_2$; S_2 **conf** S_1 and $S_2 \not\leq_{\mathcal{O}} S_1$.
5.51.4.4: S_1 **co$\not$nf** S_2 and S_1 **conf**$_Q S_2$; S_2 **conf** S_1 and S_2 **co$\not$nf**$_Q S_1$.
5.51.4.5: S_8 **co$\not$nf** S_5 and $S_8 \leq_\delta S_5$; S_3 **conf** S_1 and $S_3 \not\leq_\delta S_1$.
5.51.4.6: S_1 **co$\not$nf** S_2 and $S_1 \leq_{tr}^{Q} S_2$; S_5 **conf** S_7 and $S_5 \not\leq_{tr}^{Q} S_7$.
5.51.4.7: $S_3 \not\leq_{tr}^{Q} S_1$ and S_3 **conf**$_Q S_1$; $S_4 \leq_{tr}^{Q} S_1$ and S_4 **co$\not$nf**$_Q S_1$.
5.51.4.8: $S_8 \not\leq_{tr}^{Q} S_7$ and $S_8 \leq_\delta S_7$; $S_7 \leq_{tr}^{Q} S_5$ and $S_7 \not\leq_\delta S_5$.
5.51.4.9: S_8 **co$\not$nf**$_Q S_5$ and $S_8 \leq_\delta S_5$; S_3 **conf**$_Q$ S_1 and $S_3 \not\leq_\delta S_1$.

□

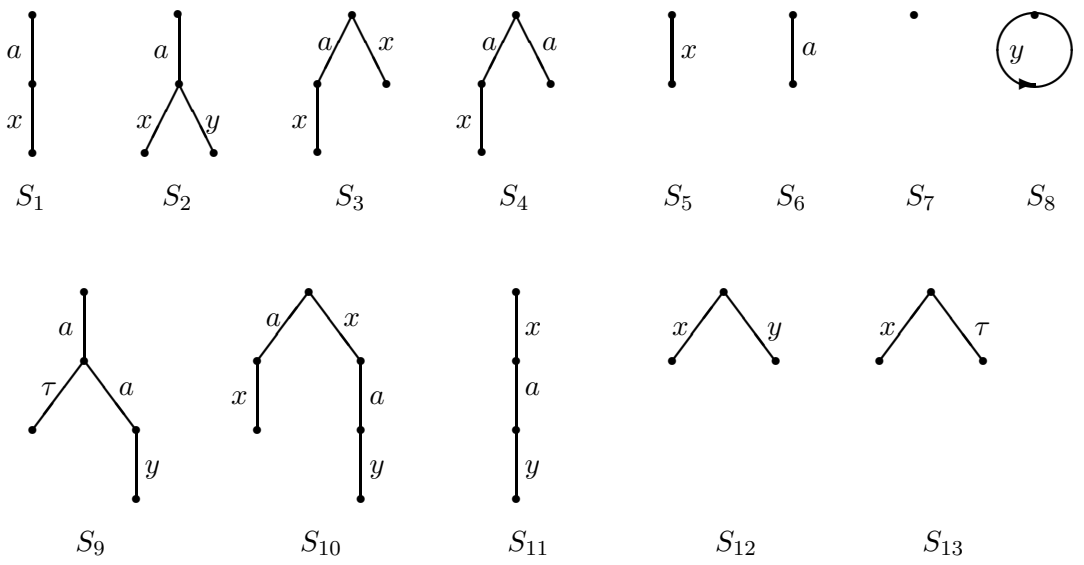


Figure B.1: Example specifications and implementations.

Proposition 5.51.5 (Finite behaviour)
For implementations with finite behaviour: $\leq_\delta = \leq_{\mathcal{O}}$

□

Proof (proposition 5.51.5)
An implementation I with finite behaviour has traces of finite length (definition 1.11.4),

implying that I deadlocks after performing a finite number of actions:

$$\forall \sigma \in traces(I) : \exists \rho \in L^* : \ I \textbf{ after } \sigma \cdot \rho \textbf{ refuses } L \qquad (1)$$

We prove $\leq_\delta \subseteq \leq_{outputs}$, from which $\leq_\delta = \leq_{\mathcal{O}}$ can be concluded, using proposition 5.51.1.4:

$$(\ \forall \sigma \in L^* : \delta_I(\sigma) \ \text{ implies } \ \delta_S(\sigma)\ ) \ \text{ implies } \ (\ \forall \sigma \in L^* : outputs_I(\sigma) \subseteq outputs_S(\sigma)\ )$$

Let $\sigma \in L^*$, $x \in outputs_I(\sigma)$, then:

	$Q_I \stackrel{\sigma \cdot x}{\Longrightarrow}$	$(*$ corollary 5.23.3 $*)$
implies	$\exists \sigma' \in traces(I) : \sigma' @ \sigma \cdot x$	$(*$ (1) $*)$
implies	$\exists \rho : \ I \textbf{ after } \sigma' \cdot \rho \textbf{ refuses } L$	$(*$ proposition 3.5.2 $*)$
implies	$I \textbf{ after } \sigma' \cdot \rho \textbf{ refuses } L_U$	$(*$ proposition 5.38 $*)$
implies	$\delta_I(\sigma' \cdot \rho)$	$(*$ premiss $*)$
implies	$\delta_S(\sigma' \cdot \rho)$	$(*$ definition 5.13.2 $*)$
implies	$Q_S \textbf{ after } \sigma' \cdot \rho \textbf{ refuses } L_U$	$(*$ definition 3.4.1 $*)$
implies	$Q_S \stackrel{\sigma' \cdot \rho}{\Longrightarrow}$	$(*$ table 1.1 $*)$
implies	$Q_S \stackrel{\sigma'}{\Longrightarrow}$	$(*$ corollary 5.23.2 $*)$
implies	$Q_S \stackrel{\sigma \cdot x}{\Longrightarrow}$	$(*$ definition 5.13.1 $*)$
implies	$x \in outputs_S(\sigma)$	

□

Proposition 5.54
asco and **aconf** are reflexive, but not transitive.

□

Proof (proposition 5.54)
Reflexivity follows immediately from the definition. For non-transitivity consider S_1, S_7, and S_9 (figure B.1): S_1 **asco** S_7, S_7 **asco** S_9, but S_1 **as$\not$co** S_9, because $\mathcal{O}_{S_1}(a \cdot a) = \{x\} \not\subseteq \mathcal{O}_{S_9}(a \cdot a) = \{\delta, y\}$. **aconf** gives exactly the same results for this example.

□

Proposition 5.55

1. $\leq_{\mathcal{O}} \ = \ \leq_{L^*} \ = \ \leq_{tr(Q_S)} \ \subset \ \leq_{tr(S)} \ \subset \ \textbf{aconf} \ \subset \ \textbf{asco}$
2. The relations **asco** and **aconf** do not contain, nor are contained in $\leq^Q_{tr}$ ($\leq_{outputs}$), $\textbf{conf}_Q$, or $\leq_\delta$.

□

Proof (proposition 5.55)

1. $\leq_{\mathcal{O}} = \leq_{L^*}$: Directly from the definitions.

   $\leq_{\mathcal{O}} = \leq_{tr(Q_S)}$: The *only-if*-part is trivial.
   The *if*-part is proved by contraposition:

$$\exists \sigma_0 \in L^* : \mathcal{O}_I(\sigma_0) \not\subseteq \mathcal{O}_S(\sigma_0) \ \text{ implies } \ \exists \sigma_1 \in traces(Q_S) : \mathcal{O}_I(\sigma_1) \not\subseteq \mathcal{O}_S(\sigma_1)$$

For $\sigma_0 \in traces(Q_S)$ we trivially choose $\sigma_1 = \sigma_0$.
For $\sigma_0 \in L^* \setminus traces(Q_S)$ we have that $\mathcal{O}_S(\sigma_0) = \emptyset$, and thus $\mathcal{O}_I(\sigma_0) \neq \emptyset$.
Thus $\sigma_0 \notin traces(Q_S), \sigma_0 \in traces(Q_I)$
implying there exist $\sigma_1, \sigma_2 \in L^*, x \in L_U$ such that:
$\sigma_0 = \sigma_1 \cdot x \cdot \sigma_2, \sigma_1 \in traces(Q_S)$,
$\sigma_1 \cdot x \notin traces(Q_S)$ and $\sigma_1 \cdot x \in traces(Q_I)$
implying $x \notin \mathcal{O}_S(\sigma_1)$ and $x \in \mathcal{O}_I(\sigma_1)$
hence for $\sigma_1 \in traces(Q_S)$ we conclude: $\mathcal{O}_I(\sigma_1) \not\subseteq \mathcal{O}_S(\sigma_1)$

$\leq_{\mathcal{O}} \subseteq \leq_{tr(S)}$: Immediately from $traces(S) \subseteq L^*$.

$\leq_{tr(S)} \subseteq$ **aconf**: Immediately from $\{\sigma' \mid \exists \sigma \in tracks(S) : \sigma' \preceq \sigma\} \subseteq traces(S)$.

aconf $\subseteq$ **asco**:
From $\{\sigma \mid \exists x \in L_U : \sigma \cdot x \in tracks(S)\} \subseteq \{\sigma' \mid \exists \sigma \in tracks(S) : \sigma' \preceq \sigma\}$.

$\leq_{\mathcal{O}} \neq \leq_{tr(S)}$: $S_1 \not\leq_{\mathcal{O}} S_7$ and $S_1 \leq_{tr(S)} S_7$.

$\leq_{tr(S)} \neq$ **aconf**: In figure 5.13: $I_2 \not\leq_{tr(S)} S_2$ and I_2 **aconf** S_2.

aconf $\neq$ **asco**: S_3 **aco$\not$nf** S_1 and S_3 **asco** S_1.

2. **aconf, asco** $\neq \leq^Q_{tr}$: S_4 **aco$\not$nf** , **as$\not$co** S_1 and $S_4 \leq^Q_{tr} S_1$;
S_1 **aconf, asco** S_7 and $S_1 \not\leq^Q_{tr} S_7$.

aconf, asco $\neq \leq_\delta$: S_8 **aco$\not$nf** , **as$\not$co** S_5 and $S_8 \leq_\delta S_5$;
S_1 **aconf, asco** S_7 and $S_1 \not\leq_\delta S_7$.

aconf, asco $\neq$ **conf**$_Q$: S_{12} **aco$\not$nf** , **as$\not$co** S_{13} and S_{12} **conf**$_Q S_{13}$;
S_{10} **aconf, asco** S_{11} and S_{10} **co$\not$nf**$_Q S_{11}$.

□

Proposition 5.58

1. $\sigma_1 @^0 \sigma_2$ iff $\sigma_1 = \sigma_2$
2. $\bigcup_{n=0}^{\infty} @^n = @$
3. $tracks^0(S) = tracks(S)$
4. $tracks^n(S) \subseteq traces(Q_S)$
5. $\bigcup_{n=0}^{\infty} tracks^n(S) = traces(Q_S)$
6. **aconf**^{0} = **aconf**
7. $\bigcap_{n=0}^{\infty}$ **aconf**^{n} $= \leq_{\mathcal{O}}$
8. $m \leq n$ implies **aconf**$^m \supseteq$ **aconf**^{n}

□

Proof (proposition 5.58)

1. For $\sigma_1, \sigma_2 \in L_I^*$, it follows immediately that $\sigma_1 @^0 \sigma_2$ iff $\sigma_1 = \sigma_2$.
For $\sigma_1 = \rho_1 \cdot x_1 \cdot \sigma_1'$, $\sigma_2 = \rho_2 \cdot x_2 \cdot \sigma_2'$, with $\rho_1, \rho_2 \in L_I^*$, $x_1, x_2 \in L_U$, $\sigma_1', \sigma_2' \in L^*$, we have $\sigma_1 @^0 \sigma_2$ iff $\exists m \leq 0, |\rho_2 \backslash \rho_1| = m$, $x_1 = x_2$, $\sigma_1' @^{0-m} (\rho_2 \backslash \rho_1) \cdot \sigma_2'$. This holds if and only if $m = 0$, hence $\rho_1 = \rho_2$, and $\sigma_1' @^0 \sigma_2'$ iff $\sigma_1 = \sigma_2$ according to the induction hypothesis. Thus $\sigma_1 = \sigma_2$.

2. $\subseteq$: Suppose $\langle \sigma_1, \sigma_2 \rangle \in \bigcup_{n=0}^{\infty} @^n$.
   This implies $\exists n : \sigma_1 @^n \sigma_2$, implying ($*$ by definition $@^n \subseteq @$ $*$) $\sigma_1 @ \sigma_2$.

   $\supseteq$: By induction; first, suppose $\sigma_1 @ \sigma_2$, with $\sigma_1, \sigma_2 \in L_I^*$:
   $\sigma_1 @ \sigma_2$ implies $\sigma_1 \preceq \sigma_2$, implies $\exists \rho \in L_I^* : \sigma_1 \cdot \rho = \sigma_2$, implies $\sigma_1 @^{|\rho|} \sigma_2$.
   Secondly, suppose $\sigma_1 @ \sigma_2$, with $\sigma_1 = \rho_1 \cdot x_1 \cdot \sigma_1'$, $\sigma_2 = \rho_2 \cdot x_2 \cdot \sigma_2'$, with $\rho_1, \rho_2 \in L_I^*$, $x_1, x_2 \in L_U$, $\sigma_1', \sigma_2' \in L^*$: $\sigma_1 @ \sigma_2$ implies $\exists \rho \in L_I^* : \rho_1 \cdot \rho = \rho_2$ and $x_1 = x_2$ and $\sigma_1' @ (\rho_2 \backslash \rho_1) \cdot \sigma_2'$.
   By induction: $\exists n : \sigma_1' @^n (\rho_2 \backslash \rho_1) \cdot \sigma_2'$. Thus $\sigma_1 @^{n+|\rho|} \sigma_2$.

3. $$\begin{array}{lcl} \mathit{tracks}^0(S) & = & \{\ \sigma \in L^* \mid \exists \sigma' \in \mathit{tracks}(S) : \sigma' @^0 \sigma\ \} \\ & = & \{\ \sigma \in L^* \mid \exists \sigma' \in \mathit{tracks}(S) : \sigma' = \sigma\ \} \\ & = & \mathit{tracks}(S) \end{array}$$

4. $$\begin{array}{ll} & \sigma \in \mathit{tracks}^n(S) \\ \text{implies} & \exists \sigma' \in \mathit{tracks}(S) : \sigma' @^n \sigma \\ \text{implies} & (* \ \sigma' \in \mathit{tracks}(S) \text{ implies } \sigma' \in \mathit{traces}(S), \\ & \quad \sigma' @^m \sigma \text{ implies } \sigma' @ \sigma \ *) \\ & \sigma \in \mathit{traces}(Q_S) \end{array}$$

5. $\subseteq$:
   $$\begin{array}{ll} & \sigma \in \bigcup_{n=0}^{\infty} \mathit{tracks}^n(S) \\ \text{implies} & \exists n \geq 0 : \sigma \in \mathit{tracks}^n(S) \\ \text{implies} & (* \text{ proposition 5.58.4 } *) \\ & \sigma \in \mathit{traces}(Q_S) \end{array}$$

   $\supseteq$:
   $$\begin{array}{ll} & \sigma \in \mathit{traces}(Q_S) \\ \text{implies} & \exists \sigma' \in \mathit{tracks}(S) : \sigma' @ \sigma \\ \text{implies} & (* \text{ proposition 5.58.2 } *) \\ & \exists n \geq 0, \exists \sigma' \in \mathit{tracks}(S) : \sigma' @^n \sigma \\ \text{implies} & \exists n \geq 0 : \sigma \in \mathit{tracks}^n(S) \\ \text{implies} & \sigma \in \bigcup_{n=0}^{\infty} \mathit{tracks}^n(S) \end{array}$$

6. $$\begin{array}{ll} & I \ \mathbf{aconf}^0 \ S \\ \text{iff} & \forall \sigma \in \mathit{lcl}_{\preceq}(\bigcup_{i=0}^{0} \ \mathit{tracks}^i(S)) : \mathcal{O}_I(\sigma') \subseteq \mathcal{O}_S(\sigma') \\ \text{iff} & \forall \sigma \in \mathit{lcl}_{\preceq}(\mathit{tracks}(S)) : \mathcal{O}_I(\sigma') \subseteq \mathcal{O}_S(\sigma') \\ \text{iff} & I \ \mathbf{aconf} \ S \end{array}$$

7. $\subseteq$:
   $$\begin{array}{ll} & I \not\leq_{\mathcal{O}} S \\ \text{implies} & I \not\leq_{tr(Q_S)} S \\ \text{implies} & \exists \sigma \in \mathit{traces}(Q_S) : \mathcal{O}_I(\sigma) \not\subseteq \mathcal{O}_S(\sigma) \\ \text{implies} & (* \text{ proposition 5.58.5 } *) \\ & \exists n : \exists \sigma \in \mathit{tracks}^n(S) : \mathcal{O}_I(\sigma) \not\subseteq \mathcal{O}_S(\sigma) \\ \text{implies} & \exists n : I \not\mathbf{aconf}^n \ S \\ \text{implies} & \langle I, S \rangle \notin \bigcap_{n=0}^{\infty} \ \mathbf{aconf}^n \end{array}$$

$\supseteq$: $I \leq_{\mathcal{O}} S$
implies $\forall \sigma \in L^* : \mathcal{O}_I(\sigma) \subseteq \mathcal{O}_S(\sigma)$
implies $\forall n, \forall \sigma \in L^* : \mathcal{O}_I(\sigma) \subseteq \mathcal{O}_S(\sigma)$
implies $(*$ taking a subset of $L^* *)$
$\forall n, \forall \sigma \in lcl_{\preceq}(\bigcup_{i=0}^{n} \ tracks^i(S)) : \mathcal{O}_I(\sigma') \subseteq \mathcal{O}_S(\sigma')$
implies $\forall n : I$ **aconf**^{n} S
implies $\langle I, S \rangle \in \bigcap_{n=0}^{\infty}$ **aconf**^{n}

8. Suppose $m \leq n$ and I **aconf**^{n} S.
This implies $\forall \sigma \in lcl_{\preceq}(\bigcup_{i=0}^{n} \ tracks^i(S)) : \mathcal{O}_I(\sigma) \subseteq \mathcal{O}_S(\sigma)$
implies $(* \bigcup_{i=0}^{m} tracks^i(S) \subseteq \bigcup_{i=0}^{n} tracks^i(S) *)$:
$\forall \sigma \in lcl_{\preceq}(\bigcup_{i=0}^{m} \ tracks^i(S)) : \mathcal{O}_I(\sigma) \subseteq \mathcal{O}_S(\sigma)$,
implies I **aconf**^{m} S.

□

B.5.5 Section 5.8 (Test Derivation)

Proposition 5.62
$\{T_S(\sigma)\}$ is a complete test suite for S with respect to $\leq_{\{\sigma\}}$.

□

Proof (proposition 5.62)
To prove: $I \leq_{\{\sigma\}} S$ iff I **a-passes**$_{\mathcal{D}}$ $T_S(\sigma)$,
which is equivalent to ((5.2), definition 5.60, proposition 3.26):
$\mathcal{O}_I(\sigma) \subseteq \mathcal{O}_S(\sigma)$ iff Q_I **sat**$_{\mathcal{C}}$ $testreqs_{\mathcal{D}}(T_S(\sigma)$.
Distinguish between $\delta_S(\sigma)$ and not $\delta_S(\sigma)$:

not $\delta_S(\sigma)$: $testreqs_{\mathcal{D}}(T_S(\sigma)) = \{$ **after** σ **must** $A \mid outputs_S(\sigma) \subseteq A \}$:
$\mathcal{O}_I(\sigma) \subseteq \mathcal{O}_S(\sigma)$
iff $(*$ not $\delta_S(\sigma) *)$
$\mathcal{O}_I(\sigma) \subseteq outputs_S(\sigma)$
iff $(*$ lemma B.3 $*)$
Q_I **after** σ **must** $outputs_S(\sigma)$
iff $(*$ proposition 3.5.3 $*)$
$\forall A \supseteq outputs_S(\sigma) : \ Q_I$ **after** σ **must** A
iff $(*$ definition 3.16, proposition 3.26 $*)$
Q_I **sat**$_{\mathcal{C}}$ $testreqs_{\mathcal{D}}(T_S(\sigma))$

$\delta_S(\sigma)$: $testreqs_{\mathcal{D}}(T_S(\sigma)) = \{$ **after** $\sigma \cdot x$ **must** $A \mid x \in L_U \backslash outputs_S(\sigma)$ and $\emptyset \subseteq A \}$:

$\mathcal{O}_I(\sigma) \subseteq \mathcal{O}_S(\sigma)$
iff $(*\ \delta_S(\sigma)\ *)$
$outputs_I(\sigma) \subseteq outputs_S(\sigma)$
iff $(*$ definition 5.13 $*)$
$\forall x \in L_U : x \notin outputs_S(\sigma)$ implies $Q_I \stackrel{\sigma\cdot x}{\Longrightarrow\!\!\!\!\!\!/}$
iff $\forall x \in L_U \backslash outputs_S(\sigma) : Q_I \stackrel{\sigma\cdot x}{\Longrightarrow\!\!\!\!\!\!/}$
iff $(*$ proposition 3.5.5 $*)$
$\forall x \in L_U \backslash outputs_S(\sigma) :\ Q_I$ **after** $\sigma\cdot x$ **must** $\emptyset$
iff $(*$ proposition 3.5.3 $*)$
$\forall x \in L_U \backslash outputs_S(\sigma), \forall A \supseteq \emptyset :\ Q_I$ **after** $\sigma\cdot x$ **must** A
iff $(*$ definition 3.16, proposition 3.26 $*)$
Q_I **sat**$_{\mathcal{C}}$ $testreqs_{\mathcal{D}}(T_S(\sigma))$

□

Theorem 5.64
$\{T_S(\sigma) \mid \sigma \in \mathcal{F}\}$ is a complete test suite for S with respect to $\leq_{\mathcal{F}}$.

□

Proof (theorem 5.64)
$I \leq_{\mathcal{F}} S$
iff $(*\ (5.2)\ *)$
$\forall \sigma \in \mathcal{F} : \mathcal{O}_I(\sigma) \subseteq \mathcal{O}_S(\sigma)$
iff $(*\ (5.2)$ for definition $\leq_{\{\sigma\}}\ *)$
$\forall \sigma \in \mathcal{F} : I \leq_{\{\sigma\}} S$
iff $(*$ proposition 5.62 $*)$
$\forall \sigma \in \mathcal{F} : I$ **a-passes**$_{\mathcal{D}}$ $T_S(\sigma)$
iff $(*$ notation 2.8 $*)$
I **a-passes**$_{\mathcal{D}}$ $\{T_S(\sigma) \mid \sigma \in \mathcal{F}\}$

□

B.5.6 Section 5.9 (Computation of Outputs and Deadlocks)

Proposition 5.68

1. $outputs_S(\sigma) \ = \ \bigcup \{\ \mu\omega(\ \langle S', \sigma' \rangle\ )\ \mid\ \langle S', \sigma' \rangle \in \mu_S(\sigma)\ \}$
2. $\delta_S(\sigma)$ iff $\exists \langle S', \sigma' \rangle \in \mu_S(\sigma) : \mu\delta(\langle S', \sigma' \rangle)$

□

Proof (proposition 5.68)

1. To prove: $x \in outputs_S(\sigma)$ iff $\exists \langle S', \sigma' \rangle \in \mu_S(\sigma) : S' \stackrel{x}{\Longrightarrow}$ and $x \in L_U$

   if: $\exists \langle S', \sigma' \rangle \in \mu_S(\sigma) : S' \stackrel{x}{\Longrightarrow}$ and $x \in L_U$
   implies $\exists S', \sigma' : [_{\epsilon\ll} S_{\ll\epsilon}] \stackrel{\sigma}{\Longrightarrow} [_{\epsilon\ll} S'_{\ll\sigma'}]$ and $S' \stackrel{x}{\Longrightarrow}$ and $x \in L_U$
   implies $\exists S', \sigma' : [_{\epsilon\ll} S_{\ll\epsilon}] \stackrel{\sigma}{\Longrightarrow} [_{\epsilon\ll} S'_{\ll\sigma'}] \stackrel{x}{\Longrightarrow}$ and $x \in L_U$
   implies $[_{\epsilon\ll} S_{\ll\epsilon}] \stackrel{\sigma\cdot x}{\Longrightarrow}$ and $x \in L_U$
   implies $x \in outputs_S(\sigma)$

only if: Analogous to the proof of proposition 5.22.3:

$$\begin{array}{ll} & x \in \mathit{outputs}_S(\sigma) \\ \text{implies} & [_\epsilon\!\ll\! S \!\ll_\epsilon] \stackrel{\sigma\cdot x}{\Longrightarrow} \\ \text{implies} & \exists S', \sigma'_i, \sigma'_u : [_\epsilon\!\ll\! S \!\ll_\epsilon] \stackrel{\sigma\cdot x}{\Longrightarrow} [_{\sigma'_u}\!\ll\! S' \!\ll_{\sigma'_i}] \end{array}$$

implying that somewhere in this derivation the following two derivation steps must occur:

$$\begin{array}{rcll} [_{\sigma_v}\!\ll\! S_1 \!\ll_{\sigma_j}] & \stackrel{\tau}{\longrightarrow} & [_{\sigma_v\cdot x}\!\ll\! S_2 \!\ll_{\sigma_j}] & (1) \\ [_{x\cdot\sigma_w}\!\ll\! S_3 \!\ll_{\sigma_k}] & \stackrel{x}{\longrightarrow} & [_{\sigma_w}\!\ll\! S_3 \!\ll_{\sigma_k}] & (2) \end{array}$$

for some $\sigma_j, \sigma_k \in L_I^*$, $\sigma_v, \sigma_w \in L_U^*$, S_1, S_2, S_3, and with $S_1 \stackrel{x}{\longrightarrow} S_2$.

Let $\sigma_a \in L^*$ label the derivation from initial state to (1), and $\sigma_b \in L^*$ the derivation from (1) to (2), then we have:

$$\begin{array}{rcl} [_\epsilon\!\ll\! S \!\ll_\epsilon] & \stackrel{\sigma_a}{\Longrightarrow} & [_{\sigma_v}\!\ll\! S_1 \!\ll_{\sigma_j}] \\ & \stackrel{\tau}{\longrightarrow} & [_{\sigma_v\cdot x}\!\ll\! S_2 \!\ll_{\sigma_j}] \\ & \stackrel{\sigma_b}{\Longrightarrow} & [_{x\cdot\sigma_w}\!\ll\! S_3 \!\ll_{\sigma_k}] \\ & \stackrel{x}{\longrightarrow} & [_{\sigma_w}\!\ll\! S_3 \!\ll_{\sigma_k}] \\ & \stackrel{\epsilon}{\Longrightarrow} & [_{\sigma'_u}\!\ll\! S' \!\ll_{\sigma'_i}] \end{array}$$

It follows that $\sigma_v = \sigma_b\lceil L_U$, $\sigma_a\cdot\sigma_b = \sigma$, and that also the following derivation is possible:

$$[_\epsilon\!\ll\! S \!\ll_\epsilon] \stackrel{\sigma_a}{\Longrightarrow} [_{\sigma_b\lceil L_U}\!\ll\! S_1 \!\ll_{\sigma_j}] \stackrel{\sigma_b}{\Longrightarrow} [_\epsilon\!\ll\! S_1 \!\ll_{\sigma_j\cdot(\sigma_b\lceil L_I)}]$$

hence: $\langle S_1, \sigma_j\cdot(\sigma_b\lceil L_I)\rangle \in \mu_S(\sigma)$ and $S_1 \stackrel{x}{\Longrightarrow}$ and $x \in L_U$.

2.

$$\begin{array}{ll} & \delta_S(\sigma) \\ \text{iff} & (*\ \text{proposition 5.41.1}\ *) \\ & \sigma \in \delta\text{-}\mathit{empty}(S)\ \text{ or }\ \sigma \in \delta\text{-}\mathit{block}(S) \\ \text{iff} & (*\ \text{definition 5.40.1,2}\ *) \\ & \quad (\ \exists\sigma' \in \mathit{traces}(S) : \sigma' \,|@|\, \sigma\ \text{ and }\ S\ \textbf{after}\ \sigma'\ \textbf{refuses}\ L_U\ ) \\ & \text{or}\ (\ \exists\sigma' \in \mathit{traces}(S), a \in L_I : \sigma'\cdot a \,@\, \sigma\ \text{and}\ \ S\ \textbf{after}\ \sigma'\ \textbf{refuses}\ \{a\}\cup L_U\ ) \\ \text{iff} & (*\ \text{definition 3.4}\ *) \\ & \quad (\ \exists\sigma', S' : S \stackrel{\sigma'}{\Longrightarrow} S'\ \text{ and }\ \sigma' \,|@|\, \sigma\ \text{ and }\ \forall x \in L_U : S' \stackrel{x}{\Longrightarrow}\!\!\!\!\!\!/\ \ )\\ & \text{or}\ (\ \exists\sigma', S', a : S \stackrel{\sigma'}{\Longrightarrow} S'\ \text{ and }\ \sigma'\cdot a \,@\, \sigma\ \text{ and }\ \forall x \in L_U\cup\{a\} : S' \stackrel{x}{\Longrightarrow}\!\!\!\!\!\!/\ \ ) \\ \text{iff} & (*\ \text{propositions 5.33.3,4, 5.22.1,2, } \sigma' \,@\, \sigma'\cdot a \,@\, \sigma,\ a\cdot\sigma'' = \sigma\backslash\backslash\sigma'\ \ *) \\ & \quad (\ \exists S' : [_\epsilon\!\ll\! S \!\ll_\epsilon] \stackrel{\sigma}{\Longrightarrow} [_\epsilon\!\ll\! S' \!\ll_\epsilon]\ \text{ and }\ \forall x \in L_U : S' \stackrel{x}{\Longrightarrow}\!\!\!\!\!\!/\ \ ) \\ & \text{or}\ (\ \exists\sigma'', S', a : [_\epsilon\!\ll\! S \!\ll_\epsilon] \stackrel{\sigma}{\Longrightarrow} [_\epsilon\!\ll\! S' \!\ll_{a\cdot\sigma''}]\ \text{ and }\ \forall x \in L_U \cup \{a\} : S' \stackrel{x}{\Longrightarrow}\!\!\!\!\!\!/\ \ ) \\ \text{iff} & \exists S', \sigma' : [_\epsilon\!\ll\! S \!\ll_\epsilon] \stackrel{\sigma}{\Longrightarrow} [_\epsilon\!\ll\! S \!\ll_{\sigma'}]\ \text{ and} \\ & (\quad (\ \sigma' = \epsilon\ \text{ and }\ \forall x \in L_U : S' \stackrel{x}{\Longrightarrow}\!\!\!\!\!\!/\ \ ) \\ & \quad \text{or}\ (\ \exists\sigma'', a : \sigma' = a\cdot\sigma''\ \text{ and }\ \forall x \in L_U \cup \{a\} : S' \stackrel{x}{\Longrightarrow}\!\!\!\!\!\!/\ \ )\ \ ) \\ \text{iff} & (*\ \text{definition 5.67}\ *) \\ & \exists\langle S', \sigma'\rangle \in \mu_S(\sigma) : \mu\delta(\langle S', \sigma'\rangle) \end{array}$$

□

Proposition 5.69

1. $\mu_S(\epsilon) \;=\; \{\; \langle S', \epsilon\rangle \;\mid\; S \stackrel{\epsilon}{\Longrightarrow} S' \;\}$
2. $\mu_S(\sigma{\cdot}a) \;=\; \{\; \langle S', \sigma'{\cdot}a\rangle \;\mid\; \langle S', \sigma'\rangle \in \mu_S(\sigma) \;\}$
   $\cup\; \{\; \langle S'', \epsilon\rangle \;\mid\; \langle S', \epsilon\rangle \in \mu_S(\sigma) \text{ and } S' \stackrel{a}{\Longrightarrow} S'' \;\}$
3. $\mu_S(\sigma{\cdot}x) \;=\; \{\; \langle S'', \sigma''\rangle \;\mid\; \exists \langle S', \sigma'\rangle \in \mu_S(\sigma), \exists \rho \preceq \sigma' : S' \stackrel{x\cdot\rho}{\Longrightarrow} S'' \text{ and } \sigma'' = \sigma'\backslash\rho \;\}$

□

Proof (proposition 5.69)

1. From definition 5.67:
   $\mu_S(\epsilon) \;=\; \{\; \langle S', \sigma'\rangle \;\mid\; [\epsilon\ll S_{\ll\epsilon}] \stackrel{\epsilon}{\Longrightarrow} [\epsilon\ll S'_{\ll\sigma'}] \;\} \;=\; \{\; \langle S', \epsilon\rangle \;\mid\; S \stackrel{\epsilon}{\Longrightarrow} S' \;\}$
2. $\subseteq$: Let $\langle S', \sigma'\rangle \in \mu_S(\sigma{\cdot}a)$
   distinguish: $\sigma' = \epsilon$ and $\exists b \in L_I, \sigma'' \in L_I^* : \sigma' = \sigma''{\cdot}b$

   $\sigma' = \epsilon$: $\langle S', \epsilon\rangle \in \mu_S(\sigma{\cdot}a)$
   implies $[\epsilon\ll S_{\ll\epsilon}] \stackrel{\sigma\cdot a}{\Longrightarrow} [\epsilon\ll S'_{\ll\epsilon}]$
   implies $\exists \sigma_j \in L_I^*, \sigma_v \in L_U^*, S_1 :$
   $[\epsilon\ll S_{\ll\epsilon}] \stackrel{\sigma}{\Longrightarrow} [\sigma_v\ll S_{1\ll\sigma_j}] \stackrel{a}{\longrightarrow} [\sigma_v\ll S_{1\ll\sigma_j\cdot a}] \stackrel{\epsilon}{\Longrightarrow} [\epsilon\ll S'_{\ll\epsilon}]$,
   hence $\sigma_v = \epsilon$ and $S_1 \stackrel{\sigma_j\cdot a}{\Longrightarrow} S'$
   implies $\exists \sigma_j \in L_I^*, S_1, S_2, S_3 : S_1 \stackrel{\sigma_j}{\Longrightarrow} S_2 \stackrel{a}{\longrightarrow} S_3 \stackrel{\epsilon}{\Longrightarrow} S'$
   implies $\exists S_2, S_3 : [\epsilon\ll S_{\ll\epsilon}] \stackrel{\sigma}{\Longrightarrow} [\epsilon\ll S_{1\ll\sigma_j}] \stackrel{\epsilon}{\Longrightarrow} [\epsilon\ll S_{2\ll\epsilon}]$
   and $S_2 \stackrel{a}{\longrightarrow} S_3 \stackrel{\epsilon}{\Longrightarrow} S'$
   implies $\exists \langle S_2, \epsilon\rangle \in \mu_S(\sigma)$ and $S_2 \stackrel{a}{\Longrightarrow} S'$

   $\sigma' = \sigma''{\cdot}b$: $\langle S', \sigma''{\cdot}b\rangle \in \mu_S(\sigma{\cdot}a)$
   implies $[\epsilon\ll S_{\ll\epsilon}] \stackrel{\sigma\cdot a}{\Longrightarrow} [\epsilon\ll S'_{\ll\sigma''\cdot b}]$
   implies $a = b$ and $[\epsilon\ll S_{\ll\epsilon}] \stackrel{\sigma}{\Longrightarrow} [\epsilon\ll S'_{\ll\sigma''}]$
   implies $\langle S', \sigma''\rangle \in \mu_S(\sigma)$

   $\supseteq$:(1): $\langle S', \sigma'{\cdot}a\rangle \in \{\langle S', \sigma'{\cdot}a\rangle \mid \langle S', \sigma'\rangle \in \mu_S(\sigma)\}$
   implies $\langle S', \sigma'\rangle \in \mu_S(\sigma)$
   implies $[\epsilon\ll S_{\ll\epsilon}] \stackrel{\sigma}{\Longrightarrow} [\epsilon\ll S'_{\ll\sigma'}]$
   implies $[\epsilon\ll S_{\ll\epsilon}] \stackrel{\sigma}{\Longrightarrow} [\epsilon\ll S'_{\ll\sigma'}] \stackrel{a}{\longrightarrow} [\epsilon\ll S'_{\ll\sigma'\cdot a}]$
   implies $\langle S', \sigma'{\cdot}a\rangle \in \mu_S(\sigma{\cdot}a)$

   (2): $\langle S'', \epsilon\rangle \in \{\; \langle S'', \epsilon\rangle \;\mid\; \langle S', \epsilon\rangle \in \mu_S(\sigma) \text{ and } S' \stackrel{a}{\Longrightarrow} S'' \;\}$
   implies $\langle S', \epsilon\rangle \in \mu_S(\sigma)$ and $S' \stackrel{a}{\Longrightarrow} S''$
   implies $[\epsilon\ll S_{\ll\epsilon}] \stackrel{\sigma}{\Longrightarrow} [\epsilon\ll S'_{\ll\epsilon}]$ and $S' \stackrel{a}{\Longrightarrow} S''$
   implies $[\epsilon\ll S_{\ll\epsilon}] \stackrel{\sigma}{\Longrightarrow} [\epsilon\ll S'_{\ll\epsilon}] \stackrel{a}{\longrightarrow} [\epsilon\ll S'_{\ll a}] \stackrel{\epsilon}{\Longrightarrow} [\epsilon\ll S''_{\ll\epsilon}]$
   implies $\langle S'', \epsilon\rangle \in \mu_S(\sigma{\cdot}a)$

3. $\subseteq$: $\langle S', \sigma' \rangle \in \mu_S(\sigma \cdot x)$

implies $[_\epsilon \ll S \ll_\epsilon] \overset{\sigma \cdot x}{\Longrightarrow} [_\epsilon \ll S' \ll_{\sigma'}]$

implies $\exists \sigma_a, \sigma_b \in L^*, \sigma_j, \sigma_k \in L_I^*, \sigma_v, \sigma_w \in L_U^*, S_1, S_2, S_3 :$

$$\begin{array}{rcl} [_\epsilon \ll S \ll_\epsilon] & \overset{\sigma_a}{\Longrightarrow} & [_{\sigma_v} \ll S_1 \ll_{\sigma_j}] \\ & \overset{\tau}{\longrightarrow} & [_{\sigma_v \cdot x} \ll S_2 \ll_{\sigma_j}] \\ & \overset{\sigma_b}{\Longrightarrow} & [_{x \cdot \sigma_w} \ll S_3 \ll_{\sigma_k}] \\ & \overset{x}{\longrightarrow} & [_{\sigma_w} \ll S_3 \ll_{\sigma_k}] \\ & \overset{\epsilon}{\Longrightarrow} & [_\epsilon \ll S' \ll_{\sigma'}] \end{array}$$

with $\sigma = \sigma_a \cdot \sigma_b,\ \sigma_b \lceil L_U = \sigma_v,\ \sigma_w = \epsilon,\ S_1 \xrightarrow{x} S_2$

implies $\exists \sigma_a, \sigma_b \in L^*, \sigma_j, \sigma_k \in L_I^*, S_1, S_2, S_3 :$

$$\begin{array}{rcl} [_\epsilon \ll S \ll_\epsilon] & \overset{\sigma_a}{\Longrightarrow} & [_{\sigma_b \lceil L_U} \ll S_1 \ll_{\sigma_j}] \\ & \overset{\tau}{\longrightarrow} & [_{(\sigma_b \lceil L_U) \cdot x} \ll S_2 \ll_{\sigma_j}] \\ & \overset{\sigma_b}{\Longrightarrow} & [_x \ll S_3 \ll_{\sigma_k}] \\ & \overset{x}{\longrightarrow} & [_\epsilon \ll S_3 \ll_{\sigma_k}] \\ & \overset{\epsilon}{\Longrightarrow} & [_\epsilon \ll S' \ll_{\sigma'}] \end{array}$$

implies $\exists \rho_1, \rho_2 \in L_I^* : S_1 \xrightarrow{x} S_2 \overset{\rho_1}{\Longrightarrow} S_3 \overset{\rho_2}{\Longrightarrow} S'$
such that $\rho_1 \cdot \rho_2 \cdot \sigma' = \sigma_j \cdot (\sigma_b \lceil L_I)$

implies $\exists \sigma_a, \sigma_b \in L^*, \sigma_j \in L_I^*, S_1 :$
$[_\epsilon \ll S \ll_\epsilon] \overset{\sigma_a}{\Longrightarrow} [_{\sigma_b \lceil L_U} \ll S_1 \ll_{\sigma_j}] \overset{\sigma_b}{\Longrightarrow} [_\epsilon \ll S_1 \ll_{\sigma_j \cdot (\sigma_b \lceil L_I)}]$
and $\rho_1 \cdot \rho_2 \preceq \sigma_j \cdot (\sigma_b \lceil L_I)$ and $S_1 \overset{x \cdot \rho_1 \cdot \rho_2}{\Longrightarrow} S'$
and $\sigma' = (\sigma_j \cdot (\sigma_b \lceil L_I)) \backslash (\rho_1 \cdot \rho_2)$

implies $\langle\, S_1, \sigma_j \cdot (\sigma_b \lceil L_I)\, \rangle \in \mu_S(\sigma)$
and $\rho_1 \cdot \rho_2 \preceq \sigma_j \cdot (\sigma_b \lceil L_I)$ and $S_1 \overset{x \cdot \rho_1 \cdot \rho_2}{\Longrightarrow} S'$
and $\sigma' = (\sigma_j \cdot (\sigma_b \lceil L_I)) \backslash (\rho_1 \cdot \rho_2)$

implies $\langle S', \sigma' \rangle$ element of the right-hand side of 5.69.3

$\supseteq$: $\langle S'', \sigma'' \rangle$ element of the right-hand side of 5.69.3

implies $\exists \langle S', \sigma' \rangle \in \mu_S(\sigma), \exists \rho \preceq \sigma' : S' \overset{x \cdot \rho}{\Longrightarrow} S''$ and $\sigma'' = \sigma \backslash \rho$

implies $\exists S', \sigma', \rho \preceq \sigma' :$
$[_\epsilon \ll S \ll_\epsilon] \overset{\sigma}{\Longrightarrow} [_\epsilon \ll S' \ll_{\sigma'}] \overset{\epsilon}{\Longrightarrow} [_x \ll S'' \ll_{\sigma' \backslash \rho}] \xrightarrow{x} [_\epsilon \ll S'' \ll_{\sigma' \backslash \rho}]$

implies $\langle S'', \sigma'' \rangle \in \mu_S(\sigma \cdot x)$

□

B.6 Chapter 6 (Test Selection)

B.6.1 Section 6.2 (A Framework for Test Selection)

Proposition 6.4
If $v : \mathcal{P}(\mathcal{L}_{FDT}) \to \mathbf{R}_{\geq 0}$ is a valuation of $\mathcal{P}(\mathcal{L}_{FDT})$, then $\overline{v} : \mathcal{P}(\mathcal{K}) \to \mathbf{R}_{\geq 0}$, defined by

$$\overline{v}(\Pi) \quad =_{def} \quad v\,(\,\{\, I \in \mathcal{L}_{FDT} \mid I \textbf{ fails } \Pi \,\}\,)$$

is a valuation of $\mathcal{P}(\mathcal{K})$.

□

Proof (proposition 6.4)

$\Pi_1 \leq_\Pi \Pi_2$
implies $(*$ definition 6.1 $*)$
$\{I \in \mathcal{L}_{FDT} \mid I \textbf{ fails } \Pi_1\} \subseteq \{I \in \mathcal{L}_{FDT} \mid I \textbf{ fails } \Pi_2\}$
implies $(*$ definition 6.3 $*)$
$v\,(\,\{I \in \mathcal{L}_{FDT} \mid I \textbf{ fails } \Pi_1\}\,) \leq v\,(\,\{I \in \mathcal{L}_{FDT} \mid I \textbf{ fails } \Pi_2\}\,)$
implies $(*$ definition $\overline{v}$ $*)$
$\overline{v}(\Pi_1) \leq \overline{v}(\Pi_2)$

□

Proposition 6.6
For all finite $E \subseteq \mathcal{L}_{FDT}$ and $w : E \to \mathbf{R}_{\geq 0}$ w_r is a valuation of $\mathcal{P}(\mathcal{L}_{FDT})$, and $\overline{w_r}$ is a valuation of $\mathcal{P}(\mathcal{K})$.

□

Proof (proposition 6.6)

$c_1 \subseteq c_2$
implies $\{e \mid e \in E \text{ and } e \in c_1\} \subseteq \{e \mid e \in E \text{ and } e \in c_2\}$
implies $(*\ w(e) \geq 0\ *)$
$\Sigma\{w(e) \mid e \in E \text{ and } e \in c_1\} \leq \Sigma\{w(e) \mid e \in E \text{ and } e \in c_2\}$
implies $(*$ definition 6.5 $*)$
$w_r(c_1) \leq w_r(c_2)$

Proposition 6.4: if w_r is a valuation of $\mathcal{P}(\mathcal{L}_{FDT})$, then $\overline{w_r}$ is a valuation of $\mathcal{P}(\mathcal{K})$.

□

Proposition 6.8
For all $w : \Xi \to \mathbf{R}_{\geq 0}$ w_p is a valuation of $\mathcal{E}(\Xi)$.

□

Proof (proposition 6.8)

$c_1 \subseteq c_2$
implies $(*\ \ c_1, c_2 \in \mathcal{E}(\Xi)\ \ *)$
$\bigcup\{\xi_i \mid \xi_i \subseteq c_1\} \subseteq \bigcup\{\xi_i \mid \xi_i \subseteq c_2\}$
implies $(*\ \ \Xi \text{ is a partition}\ \ *)$
$\{\xi_i \mid \xi_i \subseteq c_1\} \subseteq \{\xi_i \mid \xi_i \subseteq c_2\}$
implies $(*\ \ w(\xi_i) \geq 0\ \ *)$
$\Sigma\{w(\xi_i) \mid \xi_i \subseteq c_1\} \leq \Sigma\{w(\xi_i) \mid \xi_i \subseteq c_2\}$
implies $(*\ \ \text{definition 6.7.3}\ \ *)$
$w_p(c_1) \leq w_p(c_2)$

□

Proposition 6.10
For all $u : \mathcal{L}_{FDT}/\approx_{\mathcal{K}} \rightarrow \mathbf{R}_{\geq 0}$, $\overline{u_p} : \mathcal{P}(\mathcal{K}) \rightarrow \mathbf{R}_{\geq 0}$ is a valuation of $\mathcal{P}(\mathcal{K})$.

□

Proof (proposition 6.10)
First, for all Π: $\{I \mid I \textbf{ fails } \Pi\}$ is expressible in $\mathcal{L}_{FDT}/\approx_{\mathcal{K}}$, i.e. $\{I \mid I \textbf{ fails } \Pi\} \in \mathcal{E}(\mathcal{L}_{FDT}/\approx_{\mathcal{K}})$;

$$\{I \mid I \textbf{ fails } \Pi\} \ = \ \bigcup\{\xi_i \in \mathcal{L}_{FDT}/\approx_{\mathcal{K}} |\ \xi_i \subseteq \{I \mid I \textbf{ fails } \Pi\}\}$$

$I_0 \in \{I \mid I \textbf{ fails } \Pi\}$
iff $[I_0]_{\approx_{\mathcal{K}}} \subseteq \{I \mid I \textbf{ fails } \Pi\}$
iff $I_0 \in \bigcup\{\xi_i \in \mathcal{L}_{FDT}/\approx_{\mathcal{K}} |\ \xi_i \subseteq \{I \mid I \textbf{ fails } \Pi\}\}$

Now let $u : \mathcal{L}_{FDT}/\approx_{\mathcal{K}} \rightarrow \mathbf{R}_{\geq 0}$,
then $(*\ \ \text{proposition 6.8}\ \ *)$
$u_p : \mathcal{E}(\Xi) \rightarrow \mathbf{R}_{\geq 0}$ is a valuation of $\mathcal{E}(\mathcal{L}_{FDT}/\approx_{\mathcal{K}})$
then $(*\ \ \text{proposition 6.4, and for all } \Pi\text{: } \{I \mid I \textbf{ fails } \Pi\} \in \mathcal{E}(\mathcal{L}_{FDT}/\approx_{\mathcal{K}})\ \ *)$
$\overline{u_p} : \mathcal{P}(\mathcal{K}) \rightarrow \mathbf{R}_{\geq 0}$ is a valuation of $\mathcal{P}(\mathcal{K})$

□

Proposition 6.11
For all finite $E \subseteq \mathcal{L}_{FDT}$ and $w : E \rightarrow \mathbf{R}_{\geq 0}$ there exists a $u : \mathcal{L}_{FDT}/\approx_{\mathcal{K}} \rightarrow \mathbf{R}_{\geq 0}$ such that $\overline{u_p} = \overline{w_r}$.

□

Proof (proposition 6.11)
Define $u : \mathcal{L}_{FDT}/\approx_{\mathcal{K}} \rightarrow \mathbf{R}_{\geq 0}$ by putting $u(c) = \Sigma\{w(e) \mid e \in c \cap E\}$.

Then $\overline{u_p}(\Pi)$
$= \quad (*$ proposition 6.4 $*)$
$u_p(\{I \mid I \textbf{ fails } \Pi\})$
$= \quad (*$ definition 6.7.3, $\{I \mid I \textbf{ fails } \Pi\} \in \mathcal{E}(\mathcal{L}_{FDT}/\approx_{\mathcal{K}})$ $*)$
$\Sigma\{u(\xi_i) \mid \xi_i \subseteq \{I \mid I \textbf{ fails } \Pi\}, \xi_i \in \mathcal{L}_{FDT}/\approx_{\mathcal{K}}\}$
$= \quad (*$ definition u $*)$
$\Sigma\{\Sigma\{w(e) \mid e \in \xi_i \cap E\} \mid \xi_i \subseteq \{I \mid I \textbf{ fails } \Pi\}, \xi_i \in \mathcal{L}_{FDT}/\approx_{\mathcal{K}}\}$
$= \quad (*$ $\{I \mid I \textbf{ fails } \Pi\} = \bigcup\{\xi_i \mid \xi_i \subseteq \{I \mid I \textbf{ fails } \Pi\}\}$ $*)$
$\Sigma\{w(e) \mid e \in \{I \mid I \textbf{ fails } \Pi\} \cap E\}$
$= \quad (*$ definition 6.5 $*)$
$w_r(\{I \mid I \textbf{ fails } \Pi\})$
$= \quad (*$ proposition 6.4 $*)$
$\overline{w_r}(\Pi)$

□

Proposition 6.13
Let $u : \mathcal{L}_{FDT}/\approx_P \rightarrow \mathbf{R}_{\geq 0}$, then

1. $u_p : \mathcal{E}(\mathcal{L}_{FDT}/\approx_P) \rightarrow \mathbf{R}_{\geq 0}$ is a valuation of $\mathcal{E}(\mathcal{L}_{FDT}/\approx_P)$;
2. Let E be the set of test suites for which the set of detected implementations is expressible in $\mathcal{L}_{FDT}/\approx_P$: $E = \{\Pi \subseteq \mathcal{K} \mid \{I \mid I \textbf{ fails } \Pi\} \in \mathcal{E}(\mathcal{L}_{FDT}/\approx_P)\}$, then $\overline{u_p} : E \rightarrow \mathbf{R}_{\geq 0}$ is a valuation for E.

□

Proof (proposition 6.13)
Directly from propositions 6.8 and 6.4.

□

B.6.2 Section 6.5 (Test Selection by Specification Selection)

Proposition 6.17
Let $\Pi_{\mathcal{R}}$ and $\Pi_{\mathcal{R}'}$ be sound test derivations for the implementation relations $\leq_{\mathcal{R}}$ and $\leq_{\mathcal{R}'}$ respectively. If $\leq_{\mathcal{R}} \subseteq \leq'_{\mathcal{R}}$ then $\Pi_{\mathcal{R}'}$ is sound for $\leq_{\mathcal{R}}$.

□

Proof (proposition 6.17)
To prove (definition 2.9): $\forall S, I \in \mathcal{L}_{FDT} : I \leq_{\mathcal{R}} S$ implies I **passes** $\Pi_{\mathcal{R}'}(S)$:

$I \leq_{\mathcal{R}} S$
implies $I \leq_{\mathcal{R}'} S$
implies $(*$ soundness $\Pi_{\mathcal{R}'}$ for $\leq_{\mathcal{R}'}$ $*)$
I **passes** $\Pi_{\mathcal{R}'}(S)$

□

Proposition 6.19
If the test derivation $\Pi_{\mathcal{R}}$ is sound for $\leq_{\mathcal{R}}$, and $\Theta_{\mathcal{R}}$ is a selection transformation for $\leq_{\mathcal{R}}$, then the test derivation $\Pi_{\mathcal{R}} \circ \Theta_{\mathcal{R}}$ is sound for $\leq_{\mathcal{R}}$.

□

Proof (proposition 6.19)
To prove (definition 2.9): $\forall S, I \in \mathcal{L}_{FDT} : I \leq_{\mathcal{R}} S$ implies I **passes** $\Pi_{\mathcal{R}}(\Theta_{\mathcal{R}}(S))$:

$I \leq_{\mathcal{R}} S$
implies $(*\ \Theta_{\mathcal{R}}$ is a selection transformation $*)$
$I \leq_{\mathcal{R}} \Theta_{\mathcal{R}}(S)$
implies $(*$ soundness $\Pi_{\mathcal{R}}\ *)$
I **passes** $\Pi_{\mathcal{R}}(\Theta_{\mathcal{R}}(S))$

□

Proposition 6.21
Any $\Theta_{\mathbf{conf}} : \mathcal{LTS} \rightarrow \mathcal{LTS}$ satisfying S **ext** $\Theta_{\mathbf{conf}}(S)$ for all $S \in \mathcal{LTS}$, is a selection transformation for **conf**.

□

Proof (proposition 6.21)
To prove: $\forall I, S \in \mathcal{LTS} : I$ **conf** S implies I **conf** $\Theta_{\mathbf{conf}}(S)$,
which means that, given $I, S \in \mathcal{LTS}$, and I **conf** S,
it has to be proved that: $\forall \sigma \in traces(\Theta_{\mathbf{conf}}(S)), \forall A \subseteq L :$
$\Theta_{\mathbf{conf}}(S)$ **after** σ **must** A implies I **after** σ **must** A

$\sigma \in traces(\Theta_{\mathbf{conf}}(S))$ and $\Theta_{\mathbf{conf}}(S)$ **after** σ **must** A
implies $(*\ S$ **ext** $\Theta_{\mathbf{conf}}(S)\ *)$
$\sigma \in traces(S)$ and S **after** σ **must** A
implies $(*\ I$ **conf** $S\ *)$
I **after** σ **must** A

□

Proposition 6.23
For all $A \subseteq L :\ S$ **ext** $S \backslash A$

□

Proof (proposition 6.23)
Proposition 6.27 with $\alpha(\sigma) = A$ for all σ.

□

Lemma B.5
Let $\alpha : L^* \rightarrow \mathcal{P}(L)$, $\rho, \sigma \in L^*$, and define:
$\langle \alpha, \rho \rangle \sqcap \sigma \ =_{def}\ \{\, a \in L \mid \exists \rho_1, \rho_2 \in L^* : \sigma = \rho_1 \cdot a \cdot \rho_2$ and $a \in \alpha(\rho \cdot \rho_1) \,\}$.

1. If $S \backslash \langle \alpha, \rho \rangle \stackrel{\sigma}{\Longrightarrow} S'$ then $\exists S_1 : S' = S_1 \backslash \langle \alpha, \rho \cdot \sigma \rangle$
2. If $\langle \alpha, \rho \rangle \sqcap \sigma = \emptyset$ then $(\ S \backslash \langle \alpha, \rho \rangle \stackrel{\sigma}{\Longrightarrow} S' \backslash \langle \alpha, \rho \cdot \sigma \rangle$ iff $S \stackrel{\sigma}{\Longrightarrow} S'\ )$
3. if $\langle \alpha, \rho \rangle \sqcap \sigma \neq \emptyset$ then $S \backslash \langle \alpha, \rho \rangle \stackrel{\sigma}{\Longrightarrow\!\!\!\!\!\!/}$

□

Proof (lemma B.5)

1. By induction on the length of σ:

   ϵ: Directly from the fact hat the only applicable inference rule is $I3_R$.

$\sigma\cdot a$: $S\backslash\langle\alpha,\rho\rangle \stackrel{\sigma\cdot a}{\Longrightarrow} S'$
implies (∗ induction hypothesis ∗)
$\exists S_1 : S\backslash\langle\alpha,\rho\rangle \stackrel{\sigma}{\Longrightarrow} S_1\backslash\langle\alpha,\rho\cdot\sigma\rangle \stackrel{\epsilon}{\Longrightarrow} \stackrel{a}{\longrightarrow} \stackrel{\epsilon}{\Longrightarrow} S'$
implies (∗ inference rules $I2_R$ and $I3_R$ ∗)
$\exists S_1,S_2,S_3,S_4 : S\backslash\langle\alpha,\rho\rangle \stackrel{\sigma}{\Longrightarrow} S_1\backslash\langle\alpha,\rho\cdot\sigma\rangle$
$\stackrel{\epsilon}{\Longrightarrow} S_2\backslash\langle\alpha,\rho\cdot\sigma\rangle \stackrel{a}{\longrightarrow} S_3\backslash\langle\alpha,\rho\cdot\sigma\cdot a\rangle \stackrel{\epsilon}{\Longrightarrow} S_4\backslash\langle\alpha,\rho\cdot\sigma\cdot a\rangle$
implies $\exists S_4 : S\backslash\langle\alpha,\rho\rangle \stackrel{\sigma\cdot a}{\Longrightarrow} S_4\backslash\langle\alpha,\rho\cdot\sigma\cdot a\rangle$

2. By induction on the length of σ:

ϵ: Directly from the fact hat the only applicable inference rule is $I3_R$.

$\sigma\cdot a$: Let $\langle\alpha,\rho\rangle \sqcap \sigma\cdot a = \emptyset$,

then $S\backslash\langle\alpha,\rho\rangle \stackrel{\sigma\cdot a}{\Longrightarrow} S'\backslash\langle\alpha,\rho\cdot\sigma\cdot a\rangle$
iff (∗ lemma B.5.1 and induction hypothesis ∗)
$\exists S_1 : S \stackrel{\sigma}{\Longrightarrow} S_1$ and $S_1\backslash\langle\alpha,\rho\cdot\sigma\rangle \stackrel{\epsilon}{\Longrightarrow} \stackrel{a}{\longrightarrow} \stackrel{\epsilon}{\Longrightarrow} S'\backslash\langle\alpha,\rho\cdot\sigma\cdot a\rangle$
iff ($I2_R$ and $I3_R$, and $\langle\alpha,\rho\rangle \sqcap \sigma\cdot a = \emptyset$ implies $a \notin \alpha(\rho\cdot\sigma)$ ∗)
$\exists S_1 : S \stackrel{\sigma}{\Longrightarrow} S_1$ and $S_1 \stackrel{\epsilon}{\Longrightarrow} \stackrel{a}{\longrightarrow} \stackrel{\epsilon}{\Longrightarrow} S'$
iff $S \stackrel{\sigma\cdot a}{\Longrightarrow} S'$

3. $\langle\alpha,\rho\rangle \sqcap \sigma\cdot a \neq \emptyset$
implies $\exists a,\rho_1,\rho_2 : \sigma = \rho_1\cdot a\cdot\rho_2$ and $a \in \alpha(\rho\cdot\rho_1)$
implies
either $S\backslash\langle\alpha,\rho\rangle \stackrel{\rho_1}{\Longrightarrow\!\!\!\!\!\!/}$
implies $S\backslash\langle\alpha,\rho\rangle \stackrel{\sigma}{\Longrightarrow\!\!\!\!\!\!/}$
or $S\backslash\langle\alpha,\rho\rangle \stackrel{\rho_1}{\Longrightarrow} S'\backslash\langle\alpha,\rho\cdot\rho_1\rangle$
implies (∗ $a \in \alpha(\rho\cdot\rho_1)$ ∗)
$S\backslash\langle\alpha,\rho\rangle \stackrel{\rho_1}{\Longrightarrow} S'\backslash\langle\alpha,\rho\cdot\rho_1\rangle \stackrel{a}{\Longrightarrow\!\!\!\!\!\!/}$
implies $S\backslash\langle\alpha,\rho\rangle \stackrel{\sigma}{\Longrightarrow\!\!\!\!\!\!/}$ □

Proposition 6.27
For all $\alpha : L^* \to \mathcal{P}(L)$ and $\rho \in L^*$: S **ext** $S\backslash\langle\alpha,\rho\rangle$ □

Proof (proposition 6.27)

traces$(S) \supseteq$ *traces*$(S\backslash\langle\alpha,\rho\rangle)$:
$S\backslash\langle\alpha,\rho\rangle \stackrel{\sigma}{\Longrightarrow}$
implies (∗ lemma B.5.3 ∗)
$\langle\alpha,\rho\rangle \sqcap \sigma = \emptyset$
implies (∗ lemma B.5.2 ∗)
$S \stackrel{\sigma}{\Longrightarrow}$

S **conf** $S\backslash\langle\alpha,\rho\rangle$: To prove for $\sigma \in$ *traces*$(S\backslash\langle\alpha,\rho\rangle)$, $A \subseteq L$:

$\forall S'$ ($S\backslash\langle\alpha,\rho\rangle \stackrel{\sigma}{\Longrightarrow} S'\backslash\langle\alpha,\rho\cdot\sigma\rangle$ implies $\exists a \in A : S'\backslash\langle\alpha,\rho\cdot\sigma\rangle \stackrel{a}{\Longrightarrow}$)
implies $\forall S''$ ($S \stackrel{\sigma}{\Longrightarrow} S''$ implies $\exists a \in A : S'' \stackrel{a}{\Longrightarrow}$)

$S \stackrel{\sigma}{\Longrightarrow} S''$
implies (∗ lemma B.5.2 ∗)
$S\backslash\langle\alpha,\rho\rangle \stackrel{\sigma}{\Longrightarrow} S''\backslash\langle\alpha,\rho\cdot\sigma\rangle$
implies (∗ premiss ∗)
$\exists a \in A : S''\backslash\langle\alpha,\rho\cdot\sigma\rangle \stackrel{a}{\Longrightarrow}$
implies (∗ lemma B.5.3 and B.5.2 ∗)
$\exists a \in A : S'' \stackrel{a}{\Longrightarrow}$

□

Corollary 6.28
For all $S \in \mathcal{LTS}$, sound test derivations $\Pi_{\mathbf{conf}}$ for **conf**, $\alpha : L^* \to \mathcal{P}(L)$, and $\rho \in L^*$,

$$\Pi_{\mathbf{conf}}(S\backslash\langle\alpha,\rho\rangle)$$

is a sound test suite for S with respect to **conf**.

□

Proof (corollary 6.28)
(∗ proposition 6.27 ∗)
$\forall S, \alpha, \rho : S$ **ext** $S\backslash\langle\alpha,\rho\rangle$
implies (∗ proposition 6.21 ∗)
$\cdot\backslash\langle\alpha,\rho\rangle : \mathcal{LTS} \to \mathcal{LTS}$ is a selection transformation for **conf**
implies (∗ proposition 6.19 ∗)
$\Pi_{\mathcal{R}} \circ \cdot\backslash\langle\alpha,\rho\rangle$ is sound for **conf**

□

Bibliography

[Abr87] S. Abramsky. Observational equivalence as a testing equivalence. *Theoretical Computer Science*, 53(3):225–241, 1987.

[ADLU88] A.V. Aho, A.T. Dahbura, D. Lee, and M.Ü. Uyar. An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing, and Verification VIII*, pages 75–86. North-Holland, 1988.

[Ald90] R. Alderden. COOPER, the compositional construction of a canonical tester. In S.T. Vuong, editor, *FORTE'89*, pages 13–17. North-Holland, 1990.

[Bae86] J.C.M. Baeten. *Procesalgebra*. Kluwer, 1986. in Dutch.

[BAL+90] E. Brinksma, R. Alderden, R. Langerak, J. van de Lagemaat, and J. Tretmans. A formal approach to conformance testing. In J. de Meer, L. Mackert, and W. Effelsberg, editors, *Second International Workshop on Protocol Test Systems*, pages 349–363. North-Holland, 1990.

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[BDD+92] G. von Bochmann, A. Das, R. Dssouli, M. Dubuc, A. Ghedamsi, and G. Luo. Fault models in testing. In J. Kroon, R. J. Heijink, and E. Brinksma, editors, *Fourth International Workshop on Protocol Test Systems*, number C-3 in IFIP Transactions. North-Holland, 1992.

[BDZ89] G. von Bochmann, R. Dssouli, and J. R. Zhao. Trace analysis for conformance and arbitration testing. *IEEE Transactions on Software Engineering*, 15(11):1347–1356, 1989.

[Ber91] G. Bernot. Testing against formal specifications: A theoretical view. In S. Abramsky and T. S. E. Maibaum, editors, *TAPSOFT'91, Volume 2*, pages 99–119. Lecture Notes in Computer Science 494, Springer-Verlag, 1991.

[BH89] L. Brömstrup and D. Hogrefe. TESDL: Experience with generating test cases from SDL specifications. In O. Færgemand and M. M. Marques, editors, *SDL'89: The Language at Work*, pages 267–279. North-Holland, 1989.

[BK85] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.

[BKPR91] F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. The failure of failures in a paradigm for asynchronous communication. In J. C. M. Baeten and J. F. Groote, editors, *CONCUR'91*, pages 111–126. Lecture Notes in Computer Science 527, Springer-Verlag, 1991.

[Bri87] E. Brinksma. On the existence of canonical testers. Memorandum INF-87-5, University of Twente, Enschede, The Netherlands, 1987.

[Bri88] E. Brinksma. A theory for the derivation of tests. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing, and Verification VIII*, pages 63–74. North-Holland, 1988.

[Bri92] E. Brinksma. On the uniqueness of fixpoints modulo observation congruence. In *CONCUR'92*. Lecture Notes in Computer Science, Springer-Verlag, 1992.

[BS86] E. Brinksma and G. Scollo. Formal notions of implementation and conformance in LOTOS. Memorandum INF-86-13, University of Twente, 1986.

[BSS87] E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS specifications, their implementations and their tests. In G. von Bochmann and B. Sarikaya, editors, *Protocol Specification, Testing, and Verification VI*, pages 349–360. North-Holland, 1987.

[BTV91] E. Brinksma, J. Tretmans, and L. Verhaard. A framework for test selection. In B. Jonsson, J. Parrow, and B. Pehrson, editors, *Protocol Specification, Testing, and Verification XI*, pages 233–248. North-Holland, 1991.

[BU91] B. S. Bosik and M. Ü. Uyar. Finite state machine based formal methods in protocol conformance testing: From theory to implementation. *Computer Networks and ISDN Systems*, 22(1):7–33, 1991.

[CCI88] CCITT. *Specification and Description Language (SDL)*, volume Blue Book X.1 of *Recommendation Z.100*. CCITT, 1988.

[Cho78] T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978.

[Chr90a] I. Christoff. Testing equivalences and fully abstract models for probabilistic processes. In J.C.M. Baeten and J.W. Klop, editors, *CONCUR '90*, pages 126–140. Lecture Notes in Computer Science 458, Springer-Verlag, 1990.

[Chr90b] I. Christoff. *Testing Equivalences for Probabilistic Processes*. PhD thesis, Uppsala University, Sweden, 1990.

[Dal80] D. van Dalen. *Logic and Structure*. Universitext. Springer-Verlag, 1980.

[DN87] R. De Nicola. Extensional equivalences for transition systems. *Acta Informatica*, 24:211–237, 1987.

[DNH84] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[Doo91] P. Doornbosch. Test derivation for full lotos. Memorandum INF-91-51, University of Twente, Enschede, The Netherlands, 1991. Master's Thesis.

[Eer87] H. Eertink. The implementation of a test derivation algorithm. Memorandum INF-87-36, University of Twente, Enschede, The Netherlands, 1987.

[Gla90] R.J. van Glabbeek. The linear time – branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, *CONCUR '90*, pages 278–297. Lecture Notes in Computer Science 458, Springer-Verlag, 1990.

[HGD92] G.J. Holzmann, P. Godefroid, and Pirottin D. Coverage preserving reduction strategies for reachability analysis. In R.J. Linn and M.Ü. Uyar, editors, *Protocol Specification, Testing, and Verification XII*. North-Holland, 1992. To appear.

[HM85] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *JACM*, 32(1):137–161, 1985.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[ISO84] ISO. *Information Processing Systems, Open Systems Interconnection, Basic Reference Model*. International Standard IS-7498. ISO, 1984.

[ISO86] ISO. *Information Processing Systems, Open Systems Interconnection, Connection Oriented Transport Protocol Specification*. International Standard IS-8073. ISO, 1986.

[ISO89a] ISO. *Information Processing Systems, Open Systems Interconnection, Estelle - A Formal Description Technique based on an Extended State Transition Model*. International Standard IS-9074. ISO, 1989.

[ISO89b] ISO. *Information Processing Systems, Open Systems Interconnection, LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. International Standard IS-8807. ISO, 1989.

[ISO91a] ISO. *Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework*. International Standard IS-9646. ISO, 1991. CCITT X.290–X.294.

[ISO91b] ISO/IEC JTC1/SC21 N6201. *Information Retrieval, Transfer and Management for OSI, Formal Methods in Conformance Testing, working draft*. Project 1.21.54, (Arles output). ISO, June 1991.

[ISO92] ISO. *Information Processing Systems, Open Systems Interconnection, Formal Description of ISO 8073 (Classes 0,1,2,3) in LOTOS*. Technical Report TR 10024. ISO/IEC, 1992. also: Memorandum INF-92-20, University of Twente, The Netherlands.

[JJH90] He Jifeng, M.B. Josephs, and C.A.R. Hoare. A theory of synchrony and asynchrony. In *TC2 Working Conference on Programming Concepts and Methods*, 1990.

[Lan90] R. Langerak. A testing theory for LOTOS using deadlock detection. In E. Brinksma, G. Scollo, and C. A. Vissers, editors, *Protocol Specification, Testing, and Verification IX*, pages 87–98. North-Holland, 1990.

[Lan92] R. Langerak. Event structures for design and transformation in LOTOS. In K.R. Parker and G.A. Rose, editors, *FORTE'91*, number C-2 in IFIP Transactions, pages 265–280. North-Holland, 1992.

[Lar90] K. G. Larsen. Ideal specification formalism = expressivity + compositionality + decidability + testability + ... In J.C.M. Baeten and J.W. Klop, editors, *CONCUR '90*, pages 33–56. Lecture Notes in Computer Science 458, Springer-Verlag, 1990.

[Led90] G. Leduc. *On the Role of Implementation Relations in the Design of Distributed Systems using LOTOS*. PhD thesis, Université de Liège, Belgium, 1990.

[LOT92] LOTOSPHERE. The lotosphere design methodology: Basic concepts. In *Deliverable Lo/WP1/T1.1/N0045/V04*. (ESPRIT 2304), 1992.

[LS89] K.G. Larsen and A. Skou. Bisimulation through probabilistic testing. In *Proceedings Principles of Programming Languages 16*. ACM, 1989.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science 92. Springer-Verlag, 1980.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Mye79] G.L. Myers. *The Art of Software Testing*. John Wiley & Sons Inc, 1979.

[Par81] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proceedings 5th GI Conference*, pages 167–183. Lecture Notes in Computer Science 104, Springer-Verlag, 1981.

[PF90] D. H. Pitt and D. Freestone. The derivation of conformance tests from lotos specifications. *IEEE Transactions on Software Engineering*, 16(12):1337–1343, 1990.

[Phi87] I. Phillips. Refusal testing. *Theoretical Computer Science*, 50(2):241–284, 1987.

[Pnu86] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Current Trends in Concurrency*, pages 510–584. Lecture Notes in Computer Science 224, Springer-Verlag, 1986.

[Ray87] D. Rayner. OSI conformance testing. *Computer Networks and ISDN Systems*, 14:79–98, 1987.

[SL89] D.P. Sidhu and T.K. Leung. Formal methods for protocol testing: A detailed study. *IEEE Transactions on Software Engineering*, 15(4):413–426, 1989.

[Spi89] J.M. Spivey. *The Z Notation: a Reference Manual*. International Series in Computer Science. Prentice Hall, 1989.

[Sti91] C. Stirling. Modal and temporal logics. Laboratory for Foundations of Computer Science Report Series ECS-LFCS-91-157, University of Edinburgh, Edinburgh, UK, 1991.

[TKB92] J. Tretmans, P. Kars, and E. Brinksma. Protocol conformance testing : A formal perspective on ISO IS-9646. In J. Kroon, R. J. Heijink, and E. Brinksma, editors, *Fourth International Workshop on Protocol Test Systems*, number C-3 in IFIP Transactions, pages 131–142. North-Holland, 1992. Extended abstract of Memorandum INF-91-32, University of Twente, Enschede, The Netherlands, 1991.

[TL90] J. Tretmans and J. van de Lagemaat. Conformiteitstesten. Memorandum INF 90-86, University of Twente, Enschede, The Netherlands, 1990. in Dutch.

[Tre90] J. Tretmans. Test case derivation from LOTOS specifications. In S.T. Vuong, editor, *FORTE'89*, pages 345–359. North-Holland, 1990.

[Tri82] K.S. Trivedi. *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*. Prentice-Hall, 1982.

[TV92] J. Tretmans and L. Verhaard. A queue model relating synchronous and asynchronous communication. In R.J. Linn and M.Ü. Uyar, editors, *Protocol Specification, Testing, and Verification XII*, IFIP Transactions. North-Holland, 1992. To appear. Extended abstract of Memorandum INF-92-04, University of Twente, Enschede, The Netherlands, 1992, and Internal Report, TFL RR 1992-1, TFL, Hørsholm, Denmark.

[VSSB91] C.A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma. Specification styles in distributed systems design and verification. *Theoretical Computer Science*, 89:179–206, 1991.

[VSZ92] R.J. Velthuys, J.M. Schneider, and G. Zörntlein. A test derivation method based on exploiting structure information. In R.J. Linn and M.Ü. Uyar, editors, *Protocol Specification, Testing, and Verification XII*. North-Holland, 1992. To appear.

[VTKB92] L. Verhaard, J. Tretmans, P. Kars, and E. Brinksma. On asynchronous testing. In G. von Bochmann, Dssouli R., and A. Das, editors, *Fifth International Workshop on Protocol Test Systems*, IFIP Transactions. North-Holland, 1992. to appear.

[Wez90] C. D. Wezeman. The CO-OP method for compositional derivation of conformance testers. In E. Brinksma, G. Scollo, and C. A. Vissers, editors, *Protocol Specification, Testing, and Verification IX*, pages 145–158. North-Holland, 1990.

[Whi87] L. J. White. *Software Testing and Verification*, volume 26 of *Advances in Computers*. Academic Press, 1987.

Index

Samenvatting

Om tot succesvolle communicatie tussen computersystemen van verschillende leveranciers te komen worden standaard communicatieprotocollen ontwikkeld en gespecificeerd. Vervolgens zijn er implementaties van deze protocollen nodig die conformeren aan deze specificaties. Testen is een methode om de correctheid van implementaties ten opzichte van de betreffende protocolspecificatie te controleren. We spreken dan van *conformance testen*, of conformiteitstesten.

Dit proefschrift behandelt een formele aanpak van protocol conformance testen. Hierbij vindt het testen plaats op basis van een formele specificatie van het protocol. Het uiteindelijke doel is te komen tot methoden voor het (automatisch) afleiden van hanteerbare verzamelingen tests uit formele specificaties. De afgeleide tests dienen bewijsbaar correct te zijn, hetgeen wil zeggen dat ze geen correcte implementaties als foutief mogen beoordelen, en bovendien moeten ze zinvol zijn: foutieve implementaties moeten met hoge waarschijnlijkheid gedetecteerd worden. Een belangrijk aspect hierbij is een formele definitie van correctheid: wanneer conformeert een protocolimplementatie aan een formele protocolspecificatie.

Uitgangspunten voor dit proefschrift zijn de huidige, informele benadering van protocol conformance testen zoals die te vinden is in de internationale standaard ISO IS-9646 "OSI Conformance Testing Methodology and Framework", en de formalismen voor de specificatie van gedistribueerde systemen die gebaseerd zijn op gelabelde transitiesystemen en procesalgebra's. De belangrijkste aspecten van de standaard ISO IS-9646 en van de gebruikte specificatieformalismen worden geïntroduceerd in hoofdstuk 1.

In hoofdstuk 2 wordt een raamwerk voor conformance testen gepresenteerd. Hiertoe wordt een formele interpretatie gegeven aan de belangrijkste concepten van de standaard ISO IS-9646, zoals conformance voorwaarde, de definitie van conformance, testdoel, testmethode en verschillende soorten tests. Deze interpretatie leidt op natuurlijke wijze tot een formele definitie van conformance als een (preordenings-)relatie op het specificatieformalisme. Zo'n relatie wordt een *implementatierelatie* genoemd.

In hoofdstuk 3 wordt het raamwerk ingevuld met bestaande implementatierelaties voor gelabelde transitiesystemen. De relaties worden geïntroduceerd uitgaande van observaties: het gedrag van een implementatie is correct als alle observaties die een omgeving van de implementatie kan maken, verklaard kunnen worden uit het gedrag van de specificatie.

Voor één van deze implementatierelaties, de relatie **conf**, worden vervolgens in hoofdstuk 4 algoritmen ontwikkeld, waarmee tests afgeleid kunnen worden uit een specificatie die gegeven is als gelabeld transitiesysteem. Met deze algoritmen kunnen verzamelingen tests afgeleid worden, die volledig en correct zijn: een implementatie is correct volgens **conf** dan en slechts dan als alle tests succesvol zijn. De testafleidingsalgoritmen werken op gelabelde transitiesystemen, en kunnen derhalve in principe ook gebruikt worden om tests af te leiden uit een gedragsexpressie in een formele taal waarvan de semantiek gegeven is als een gelabeld transitiesysteem. Een probleem ontstaat wanneer dit gelabelde transitiesysteem oneindig is in vertakkingsgraad of aantal toestanden. Deze oneindigheid maakt het onmogelijk de algoritmen te implementeren. Derhalve worden de algoritmen getransformeerd tot algoritmen die toepasbaar zijn op gedragsexpressies. Voor een simpele klasse van gedragsexpressies worden regels gegeven waarmee de tests compositioneel afgeleid kunnen worden uit deze gedragsexpressies.

De implementatierelaties van hoofdstuk 3 en de testafleidingsalgoritmen van hoofdstuk 4 gaan er vanuit dat de tester en de implementatie synchroon met elkaar kunnen communiceren, zoals dat gemodelleerd kan worden door de parallelle synchronisatieoperator op transitiesystemen. Hoofdstuk 5 laat zien dat deze theorie niet toepasbaar is wanneer de tester en de implementatie via een FIFO-buffer communiceren. Tests die afgeleid zijn voor synchrone communicatie kunnen niet alle foutieve implementaties detecteren, terwijl correcte implementaties ten onrechte als foutief herkend worden. Om dit aan te tonen wordt asynchrone communicatie tussen de tester en de implementatie geformaliseerd met behulp van een *queue-operator* op gelabelde transitiesystemen. Deze queue-operator modelleert twee FIFO-buffers, één voor invoer en één voor uitvoer. Een systeem dat asynchroon communiceert met zijn omgeving wordt een *queue-context* genoemd. Gedrag van queue-contexten blijkt te karakteriseren door twee verzamelingen van reeksen van acties (*traces*): de reeksen die een queue-context kan uitvoeren, en de reeksen die leiden tot een toestand waarbij geen uitvoer mogelijk is. Implementatierelaties gebaseerd op asynchrone communicatie worden afgeleid, en een aanzet tot testafleidingsalgoritmen voor deze relaties wordt gegeven. De afgeleide tests worden vertaald naar TTCN, de standaard notatie voor tests uit de standaard ISO IS-9646. Omdat TTCN een communicatie-mechanisme heeft dat gebruik maakt van FIFO-buffers is deze vertaling zinvol, in tegenstelling tot een vertaling van synchrone tests.

Met de testafleidingsalgoritmen uit de hoofdstukken 4 en 5 kunnen grote hoeveelheden tests gegenereerd worden. Het reduceren van de hoeveelheid tests tot een economisch en praktisch hanteerbare hoeveelheid wordt *testselectie* genoemd. In hoofdstuk 6 wordt een raamwerk voor testselectie bestudeerd, uitgaande van het toekennen van waarden en kosten aan verzamelingen tests. De waarde wordt gerelateerd aan de fout-detecterende kracht van een verzameling tests. Het raamwerk, dat onafhankelijk is van een specificatieformalisme, is uitgewerkt voor gelabelde transitiesystemen. Bovendien wordt voor transitiesystemen een techniek voor testselectie gebaseerd op specificatieselectie behandeld. In plaats van het selecteren van tests uit een te grote verzameling van tests wordt de specificatie getransformeerd, zodat de tests afgeleid van de getransformeerde,

partiële specificatie een selectie vormen van de tests afgeleid van de oorspronkelijke specificatie.

In het laatste hoofdstuk, hoofdstuk 7, worden naast de conclusies enkele onderwerpen voor verder onderzoek genoemd, zoals de relatie tussen testen en verificatie, asynchrone communicatie met andere contexten, het uitbreiden van de testselectiemethoden voor realistische specificaties, de integratie van waarden en kosten, en het testbaar ontwerpen van protocollen. Tenslotte wordt opgemerkt, dat de bruikbaarheid en de tekortkomingen van alle gepresenteerde ideeën gevalideerd moeten worden door toepassing op het testen van realistische protocollen.

Curriculum Vitae

27 augustus 1962:	geboren te Hengelo Ov.
1974 – 1980:	Gymnasium β Scholengemeenschap Bataafse Kamp Hengelo Ov.
1980 – 1986:	Studie Elektrotechniek Specialisatie Informatica Universiteit Twente
1986 – 1992:	Medewerker onderzoek Vakgroep Tele-Informatica en Open Systemen Faculteit der Informatica Universiteit Twente